AF
IFW

TRANSMITTAL FORM

(to be used for all correspondence after initial filing)

Application Number	09/882,174
Filing Date	06/14/2001
First Named Inventor	William K. Bodin
Art Unit	2154
Examiner Name	Patel, Haresh N.

Total Number of Pages in This Submission	35	Attorney Docket Number AUS920010463US1

ENCLOSURES   (*Check all that apply*)

- [ ] Fee Transmittal Form
  - [ ] Fee Attached
- [ ] Amendment/Reply
  - [ ] After Final
  - [ ] Affidavits/declaration(s)
- [ ] Extension of Time Request
- [ ] Express Abandonment Request
- [ ] Information Disclosure Statement
- [ ] Certified Copy of Priority Document(s)
- [ ] Response to Missing Parts/ Incomplete Application
  - [ ] Response to Missing Parts under 37 CFR 1.52 or 1.53

- [ ] Drawing(s)
- [ ] Licensing-related Papers
- [ ] Petition
- [ ] Petition to Convert to a Provisional Application
- [ ] Power of Attorney, Revocation Change of Correspondence Address
- [ ] Terminal Disclaimer
- [ ] Request for Refund
- [ ] CD, Number of CD(s) _____

- [ ] After Allowance communication to Technology Center (TC)
- [ ] Appeal Communication to Board of Appeals and Interferences
- [x] Appeal Communication to TC **(Appeal Notice, Brief, Reply Brief)**
- [ ] Proprietary Information
- [ ] Status Letter
- [x] Other Enclosure(s) (please Identify below):

Appeal Brief (35 pages); 7 References; and Postcard

Remarks

The Comissioner is authorized to charge or credit Deposit Account No. 09-0447.

SIGNATURE OF APPLICANT, ATTORNEY, OR AGENT

Firm or Individual name	John Biggers Reg. No. 44,537
Signature	*John Biggers*
Date	September 21, 2005

CERTIFICATE OF TRANSMISSION/MAILING

I hereby certify that this correspondence is being facsimile transmitted to the USPTO or deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on the date shown below.

Typed or printed name	Catherine Berglund		
Signature	*Catherine Berglund*	Date	September 21, 2005

If you need assistance in completing the form, call 1-800-PTO-9199 and select option 2.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
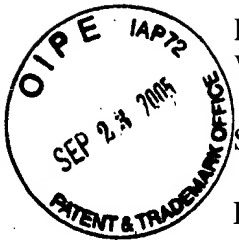
In re Application of:	§
William Kress Bodin, *et al.*	§ Group Art Unit: 2154
	§
Serial No.: 09/882,174	§
	§ Examiner: Patel, Haresh N.
Filed: 06/14/2001	§
	§ Atty Docket No.: AUS920010463US1
	§
Title: Email Routing According to Email	§
Content	§
	§

Mail Stop: Appeal Brief-Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, Virginia 22313-1450

APPEAL BRIEF

Honorable Commissioner:

This is an Appeal Brief filed pursuant to 37 CFR § 41.37 in response to the Final Office
Action of May 19, 2005 ("Final Office Action"), and pursuant to the Notice of Appeal
filed July 21, 2005.

REAL PARTY IN INTEREST

The real party in interest is the patent assignee, International Business Machines
Corporation ("IBM"), a New York corporation having a place of business at Armonk,
New York 10504.

RELATED APPEALS AND INTERFERENCES

There are no related appeals or interferences.

STATUS OF CLAIMS

Claims 1-66 are pending in the case. All pending claims are on appeal.

STATUS OF AMENDMENTS

No amendments were submitted after final rejection. The claims as currently presented are included in the Appendix of Claims that accompanies this Appeal Brief.

SUMMARY OF CLAIMED SUBJECT MATTER

Applicants provide the following concise summary of the claimed subject matter according to 37 CFR§ 41.37(c)(1)(vii), including references to specification by page and line number and to the drawing(s) if any, by reference characters.

Methods, systems, and computer program products are provided for email administration implemented in a transcoding gateway (reference 100 at Figure 2), the transcoding gateway including client device records (reference 220 at Figure 2) stored in computer memory (Lines 8-9, Page 13), each client device record (reference 250 at Figure 2) representing a client device (reference 102 at Figure 2), each client device record including a mailbox address field (reference 222 at Figure 2), an internet address field (reference 226 at Figure 2), a digital file format code field (reference 224 at Figure 2), and a path name field (reference 228 at Figure 3). The transcoding gateway further includes at least one file system, and each file system further includes file system storage locations (reference 310 at Figure 3), each file system storage location having a path name (reference 311 at Figure 3). Typical embodiments include receiving (reference 202 at Figure 2) in the transcoding gateway an email message (reference 204 at Figure 2), the

email message including at least one destination mailbox address (reference 206 at Figure 2) and at least one digital object (reference 208 at Figure 2); transcoding (reference 210 at Figure 3) the digital object into a digital file (reference 212 at Figure 3) having a digital format (reference 214 at Figure 3) and a file name (Line 23, Page 13); and downloading (reference 216 at Figure 2) the digital file to a destination client device (reference 230 at Figure 2) at an internet address (reference 218 at Figure 2) recorded in an internet address field (reference 226 at Figure 2) of a client device record (reference 220 at Figure 2). In typical embodiments, the client device record further has recorded in the client device record's mailbox address field (reference 222 at Figure 2), a mailbox address identical to the destination mailbox address (reference 206 at Figure 2) of the email message, and, recorded in the client device record's digital file format code field (reference 224 at Figure 2), a digital file format code indicating that the client device represented by the client device record is capable of receiving the digital format (reference 214 at Figure 2) of the digital file (reference 216 at Figure 2).

All such references to the specification identify descriptions and discussions that are part of the detailed descriptions of exemplary embodiments of the present invention in the present application. Such descriptions and discussions are not limitations of the claims in the present application. The only limitations of the claims are set forth in the claims themselves.

GROUNDS OF REJECTION

Claims 1-10, 13, 15-18, 23-32, 35, 37-40, 45-54, 57, and 59-62 stand rejected under 35 U.S.C § 103(a) as unpatentable over Application Server Solution Guide, Enterprise Edition: Getting Started, Nusbaum, et al., May 2000, pages 1-45, 416-434 (hereafter 'Nusbaum'); in view of Java Media Framework API Guide, JMP 2.0 FCS, November 19, 1999, Sun Microsystems, page 1-66, 109-135, 173-178 (hereafter 'Sun1'); in further view of JavaMail API Design Specification, Version1.1, Sun Microsystems, August 1998, pages 1-21, 41-50, 55-60 (hereafter 'Sun2'). Claims 11-12, 14, 33-34, 36, 55-56 and 58 stand rejected under 35 U.S.C § 103(a) as unpatentable over Nusbaum in view of Sun1

and Sun2 in further view of Carter, *et al.* (U.S. Publication No. 2002/0105545). Claims 19, 41, and 63 stand rejected under 35 U.S.C § 103(a) as unpatentable over Nusbaum in view of Sun1 and Sun2 in further view of Wenocur, *et al.* (U.S. Application No. 2003/0009694). Claims 20-22, 42-44, and 64-66 stand rejected under 35 U.S.C § 103(a) as unpatentable over Nusbaum in view of Sun1 and Sun2 in further view of Killcommons, *et al.* (U.S. Patent No. 6,424,996).

ARGUMENT

REJECTIONS FOR CLAIMS 1-66 BASED ON OBVIOUSNESS
UNDER 35 U.S.C § 103(a) ARE IMPROPER

Claims 1-10, 13, 15-18, 23-32, 35, 37-40, 45-54, 57, 59-62 stand rejected under 35 U.S.C § 103(a) as unpatentable over Nusbaum in view of Sun1 in further view of Sun2. Claims 11-12, 14, 33-34, 36, 55-56 and 58 stand rejected under 35 U.S.C § 103(a) as unpatentable over Nusbaum in view of Sun1 and Sun2 in further view of Carter. Claims 19, 41, and 63 stand rejected under 35 U.S.C § 103(a) as unpatentable over Nusbaum in view of Sun1 and Sun2 in further view of Wenocur. Claims 20-22, 42-44, and 64-66 stand rejected under 35 U.S.C § 103(a) as unpatentable over Nusbaum in view of Sun1 and Sun2 in further view of Killcommons.

In rejecting claims 1-66, the Final Office Action at pages 6 and 7 incorporates the arguments of the First Office Action of September 24, 2004 ("First Office Action"). Applicants' filed a complete response to the First Office Action on December 17, 2004 ("Response to First Office Action") demonstrating that the proposed references in the First Office Action could not possibly make obvious claims 1-66 within the meaning of 35 U.S.C. § 103(a). Applicants' therefore respond to the rejection of claims 1-66 in the Final Office Action with the arguments from Applicants' Response to First Office Action. As explained in detail below, applicants respectfully traverse the rejections of the present claims under 35 USC § 103(a).

To establish a prima facie case of obviousness, three elements must be proven by the Examiner. MPEP § 2142. The first element of a prima facie case of obviousness under 35 U.S.C. § 103 is that there must be a suggestion or motivation to modify or to combine Nusbaum, Sun1 and Sun2. *In re Vaeck*, 947 F.2d 488, 493, 20 USPQ2d 1438, 1442 (Fed. Cir. 1991). The second element of a prima facie case of obviousness under 35 U.S.C. § 103 is that there must be a reasonable expectation of success in the proposed modification or the proposed combination of Nusbaum, Sun1 and Sun2. *In re Merck & Co., Inc.*, 800

F.2d 1091, 1097, 231 USPQ 375, 379 (Fed. Cir. 1986). The third element of a prima facie case of obviousness under 35 U.S.C. § 103 is that the proposed modification or the proposed combination of Nusbaum, Sun1 and Sun2 must teach or suggest all of applicants' claim limitations. *In re Royka*, 490 F.2d 981, 985, 180 USPQ 580, 583 (CCPA 1974). As demonstrated below, neither the modification nor the combination of Nusbaum, Sun1, and Sun2 establishes a prima facie case of obviousness. The rejection of claims 1-10, 13, 15-18, 23-32, 35, 37-40, 45-54, 57, 59-62 should therefore be withdrawn and the case should be allowed.

<div align="center">

The Cited References Set Forth No Suggestion to
Modify or Combine Nusbaum, Sun1, and Sun2

</div>

To establish a prima facie case of obviousness, there must be a suggestion or motivation to modify or combine Nusbaum, Sun1, and Sun2. *In re Vaeck*, 947 F.2d 488, 493, 20 USPQ2d 1438, 1442 (Fed. Cir. 1991). "The mere fact that references can be combined or modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination." *In re Mills*, 916 F.2d 680, 16 USPQ2d 1430 (Fed. Cir. 1990). The Examiner has not pointed to any disclosure in Nusbaum, Sun1, or Sun2 suggesting the desirability of the combination. Moreover, there is no possibility whatsoever that the Examiner could ever point to any disclosure in Nusbaum, Sun1, or Sun2 suggesting the desirability of the combination. Nusbaum in fact makes no mention whatsoever of transcoding, makes no pertinent mention of email, and therefore could not possibly suggest the desirability of the combination. In addition, no such suggestion occurs in either Sun1 or Sun2. Absent such a showing of desirability, the Examiner has impermissibly used "hindsight" occasioned by applicants' own teaching to reject the claims. *In re Surko*, 11 F.3d 887, 42 U.S.P.Q.2d 1476 (Fed. Cir. 1997); *In re Vaeck*, 947 F.2d 488m 20 U.S.P.Q.2d 1438 (Fed. Cir. 1991); *In re Gorman*, 933 F.2d 982, 986, 18 U.S.P.Q.2d 1885, 1888 (Fed. Cir. 1991); *In re Bond*, 910 F.2d 831, 15 U.S.P.Q.2d 1566 (Fed. Cir. 1990); *In re Laskowski*, 871 F,.2d 115, 117, 10 U.S.P.Q.2d 1397, 1398 (Fed. Cir. 1989). The proposed combination of Nusbaum, Sun1, and Sun2

therefore cannot possibly establish a prima facie case of obviousness. The objection should be withdrawn, and the case should be allowed.

<u>There is No Reasonable Expectation Of Success in the</u>
<u>Proposed Combination of Nusbaum, Sun1, and Sun2</u>

To establish a prima facie case of obviousness, there must be a reasonable expectation of success in the proposed combination of Nusbaum, Sun1 and Sun2. *In re Merck & Co., Inc.,* 800 F.2d 1091, 1097, 231 USPQ 375, 379 (Fed. Cir. 1986). The Examiner has not pointed to any disclosure in Nusbaum, Sun1, and Sun2 suggesting any expectation of success. Absent such a showing of an expectation of success, the Examiner has failed to meet one of the three basic elements of a prima facie case of obviousness.

There can be no reasonable expectation of success in a proposed combination if the proposed combination changes the principles of operation of Nusbaum, Sun1, and Sun2. *In re Ratti,* 270 F.2d 810, 123 USPQ 349 (CCPA 1959). The Final Office Action at page 6 citing by reference the First Office Action at page 8 states:

> As per claims 1, 23, 45, Nusbaum teaches a method, system and a
> software product to implement an email administration in a transcoding
> gateway (e.g., use of application server, title), the transcoding gateway
> comprising client device records stored in computer memory (e.g., Java
> Server pages containing client information, section 1.4, page 13, each
> client device record representing a client device, (e.g., Java Server pages
> containing client information, section 1.4, page 13) ...

What Nusbaum actually states in section 1.4, page 13 is:

> JavaServer Pages (JSP) technology provides developers with an easy and
> powerful way to build Web pages with dynamic content. JSPs
> dynamically generate HTML, eXtensible Markup Language (XML), ...

That is, Nusbaum teaches a kind of dynamic web page technology known as Java Server Pages or 'JSP.' As stated in Nusbaum, "JavaServer Pages (JSP) technology provides developers with an easy and powerful way to build Web pages with dynamic content." Nusbaum, section 1.4, page 13.

Sun1 and Sun2 taken together teach some kind of transcoding and some kind of email:

> "Transcoding is the process of converting each track of media data from one input to another." Sun1, page 33.

> "Application developers who need to 'mail-enable' their applications." Sun2, page 1.

As described in Nusbaum, dynamic web page technology is methods and systems for building server pages on the fly. Clearly dynamic web pages generally and JSPs in particular, that is, "Web pages with dynamic content," are not email. Email, transcoding or not, in fact is not and cannot be implemented as part of dynamic web page technology, that is, for building web pages dynamically, without changing the principals of operation of the dynamic web page technology.

For further explanation, applicants note with respect that dynamic web page technology as described in Nusbaum takes as its inputs HTTP REQUEST and HTTP POST messages bearing query data representing parameters whose varying values affect dynamism among web page structures. Dynamic web page technology as described in Nusbaum produces as its outputs HTTP RESPONSE messages. This dynamic web page functionality as described in Nusbaum includes neither transcoding functionality nor email functionality, and transcoding and email functionality cannot be added to it without changing its principals of operation. It is pertinent to note in support of this conclusion that neither the word "transcode" nor the word "email" occurs ever, not even once, anywhere in Nusbaum. There cannot possibly be to one of ordinary skill in the art at the

8

time of the invention a reasonable expectation of success with the combination of Nusbaum, Sun1, and Sun2. The proposed combination of Nusbaum, Sun1, and Sun2 therefore cannot support a prima facie case of obviousness. The rejection should be withdrawn, and the case should be allowed.

<div align="center">

Nusbaum Teaches Away From the

Claims of the Present Application

</div>

Turning now to the substance of Nusbaum, Nusbaum actually teaches away from the current application. Teaching away from the claims is a *per se* demonstration of lack of prima facie obviousness. *In re Dow Chemical Co.,* 837 F.2d 469, 5 U.S.P.Q.2d 1529 (Fed. Cir. 1988); *In re Fine,* 837 F.2d 1071, 5 U.S.P.Q.2d 1596 (Fed. Cir. 1988); *In re Neilson,* 816 F.2d 1567, 2 U.S.P.Q.2d 1525 (Fed. Cir. 1987). Nusbaum discloses dynamic web page technology with no mention of transcoding, gateways, or email. Clearly there would be no impulse on the part of developers of dynamic web page technology to incorporate transcoding gateways or email into dynamic page technology. By effecting dynamic web page technology alone, with no hint or suggestion that transcoding gateways or pertinent email technology might even exist, Nusbaum teaches directly away from the combination with Sun1 and Sun 2 proposed in the Office Action. Nusbaum teaches a kind of dynamic web page technology: "JavaServer Pages (JSP) technology provides developers with an easy and powerful way to build Web pages with dynamic content." Nusbaum, Section 1.4, page 13. Nusbaum does not teach a method of email administration as claimed in the present application. As such, Nusbaum teaches away from applicants' claims. Because Nusbaum teaches away from the applicants claims, the proposed modification of Nusbaum with Sun1 and Sun2 cannot support a prima facie case of obviousness. The rejection of applicants' claims should be withdrawn and the case should be allowed.

<div align="center">

9

</div>

<u>Nusbaum, Sun1, and Sun2 Do Not Teach</u>
<u>Each and Every Element of the Claim</u>

To establish a prima facie case of obviousness, the proposed combination of Nusbaum, Sun1, and Sun2 must disclose all of applicants' claim limitations. *In re Royka*, 490F.2d 981, 985, 180 USPQ 580, 583 (CCPA 1974).

The First Office Action states that Sun2 at page 55 teaches the claimed elements of "receiving an email message," "the email message having destination mailbox address and object," "email information having mailbox address field," "a mailbox address identical to the destination mailbox address of the email message," "an internet address field," "a file format code field: and "a path name field." In fact, Sun2, which at page 55 merely refers to MIME parts and MIME RFCs, makes no mention whatsoever of the claimed data elements. The fact that Sun2 makes general references to MIME parts in email messages or to MIME RFCs is completely insufficient to anticipate or suggest claim elements in the present application. This ground of rejection should be withdrawn.

<u>Nusbaum Cannot be a Reference Against the Claims of the Present</u>
<u>Application Because Nusbaum Represents Nonanalogous Art</u>

Nusbaum cannot be a reference against the claims of the present application because Nusbaum represents nonanalogous art within the meaning of *In Re Horn*, *Clay*, and *Oeitker*. *In re Horn*, 203 USPQ 969 (CCPA 1979), *In re Clay*, 966 F.2d 656, 23 USPQ2d 1058 (Fed. Cir. 1992), *In re Oeticker*, 977 F.2d 1443, 24 USPQ2d 1443 (Fed. Cir. 1992). The field of the inventors' effort in this case is routing email according to its content. The present application claims, among other things, receiving an email message in a transcoding gateway, transcoding into a digital file a digital object included in the email message, and downloading the digital file to a destination client device. The field of Nusbaum is dynamic web pages for the World Wide Web – which clearly has nothing

10

to do with the technical field of the present application. Nusbaum therefore is not within the field of the inventor's endeavor in this case.

Because Nusbaum is not within the field of the inventor's endeavor in this case, there can be no basis for believing that Nusbaum as a reference would have been considered by one skilled in the particular art working on the relevant problem to which this invention pertains. That is, there would be no reason for an inventor concerned with transcoding gateways for email to search for art regarding dynamic generation of web pages. The two simply have nothing to do with one another. Nusbaum as a reference therefore is not reasonably pertinent to the particular problem with which the inventors were involved in the present case and is not available as a reference against the present application. Applicants respectfully propose that for this reason alone the rejection of the present claims should be withdrawn, and the claims should be allowed.

<div align="center">Conclusion</div>

All claims in the present case stand rejected under 35 U.S.C § 103(a). Independent claims 1, 23, and 45 stand rejected under 35 U.S.C § 103(a) over Nusbaum in view of Sun1 further in view of Sun2. The combination of Nusbaum, Sun1, and Sun2 fails to establish a prima face case of obviousness. The applicants have demonstrated that it is incorrect to reject the independent claims 1, 23, and 45 under 35 U.S.C § 103(a). The applicants respectfully propose that all the dependent claims in the present case stand because the independent claims 1, 23, and 45 stand. The rejection of all the claims 1 - 66 should therefore be withdrawn, and the claims should be allowed. Reconsideration of claims 1- 66 in light of the present remarks is respectfully requested.

APPPLICANTS' RESPONSE TO EXAMINER'S RESPONSE TO APPLICANTS' RESPONSE TO THE FIRST OFFICE ACTION DATED SEPTEMBER 24, 2004

The Final Office Action responds to Applicants' Response to First Office Action with the following arguments. First, Nusbaum, Sun1, and Sun2 are properly combined for an

obviousness rejection under 35 U.S.C. § 103. Second, Nusbaum, Sun1, and Sun2 set forth a suggestion or motivation to combine Nusbaum, Sun1, and Sun2. Third, there is a reasonable expectation of success in the combination of Nusbaum, Sun1, and Sun2. Fourth, Nusbaum is analogous art because it is in the field of Applicants' endeavor or reasonably pertinent to the particular problem with which the Applicants were concerned. Finally, Sun2 teaches the limitations cited in the First Office Action by proposing new references in Sun2. Applicants respectfully traverse each rejection of claims 1-66 under 35 U.S.C. § 102(e) and respond below in detail to the new arguments set forth in the Final Office Action.

<u>Nusbaum, Sun1, And Sun2 Are Not Properly Combined</u>
<u>For An Obviousness Rejection Under 35 U.S.C. § 103</u>

The Final Office Action at page 2 argues that Nusbaum, Sun1, and Sun2 are properly combined for an obviousness rejection under 35 U.S.C. § 103 on the grounds that:

> The cited references, Nusbaum, Sun1, and Sun2 teach a method, a system, a computer software product for email administration, all what the applicant is trying to accomplish, as per the claimed invention. Nusbaum discloses use of a transcoding gateway/server for software administration (e.g., figure 8, page 8). Sun1 discloses well known concept of transcoding (e.g., pages 4, 6 and 33). Sun2 discloses handling of email messages (e.g., page 1).

As explained in detail above in this Brief, the cited references in fact do not "teach a method, a system, a computer software product for email administration, all what the applicant is trying to accomplish, as per the claimed invention." What Nusbaum teaches generally is a kind of dynamic web page technology known as Java Server Pages. What Nusbaum teaches specifically at Figure 8 is a sample Java Server Page file and the Java code generated from that file. What Sun1 teaches generally is the Java Media API, an application programming interface for deliver of time-based media. What Sun1 teaches

specifically at pages 4, 6, and 33 is a general description of streaming media and content types, common audio formats, and a general description of media player operations. What Sun2 teaches generally is the JavaMail API, a set of abstract classes defining objects that comprise a mail system. What Sun2 teaches specifically at page 1 is a brief general introduction to the JavaMail API and a description of the target audience of the document. Not only do the cited portions of the references fail to disclose or suggest elements of the present claims, even if they did so, it would still be improper to arbitrarily pick and choose with massive hindsight elements of method and system from Java Server Pages, the Java Media API, and the JavaMail API and use them as a basis to conclude the present claims invalid for obviousness. For these reasons, Applicants continue to assert that the cited references are not properly combined in this case.

The Final Office Action at page 2 and top page 3, citing *In re Keller*, 642 F.2d 413, 425, 208 USPQ 871, 881 (CCPA 1981) and *In re Young*, 927 F.2d 588, 591 18 USPQ2d 1089, 1091 (Fed. Cir. 1991), continues its argument with:

> All cited references support the claimed method, a system and a computer software product. Also, The test for obviousness is not whether the features of a secondary reference may be bodily incorporated into the structure of a primary reference. It is also not that the claimed invention must be expressly suggested in any one or all of the references.

In this way, the Final Office Action implicitly argues that there is no need for the Examiner to demonstrate that the references provide motivation or suggestion to combine or that there is any reasonable expectation of success in combining the references so long as elements of the present claims are disclosed in the references.

As the Commissioner is well aware, however, such is not the law. The mere fact that references can be combined or modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination. In re Mills, 916 F.2d 680, 16 USPQ2d 1430 (Fed. Cir. 1990). In fact, the requirement of a prima

facie case of obviousness places a burden on the examiner to provide some suggestion of the desirability of doing what the inventor has done. "To support the conclusion that the claimed invention is directed to obvious subject matter, either the references must expressly or impliedly suggest the claimed invention or the examiner must present a convincing line of reasoning as to why the artisan would have found the claimed invention to have been obvious in light of the teachings of the references." Ex parte Clapp, 227 USPQ 972, 973 (Bd. Pat. App. & Inter. 1985). When the motivation to combine the teachings of the references is not immediately apparent, it is the duty of the examiner to explain why the combination of the teachings is proper. Ex parte Skinner, 2 USPQ2d 1788 (Bd. Pat. App. & Inter. 1986); MPEP § 2142.

The Final Office Action merely continues the practice begun in the First Office Action of pointing to elements of method and system in its cited references and stating that they are the same things claimed in the present patent application. The Final Office Action makes no substantive attempt whatsoever to present a prima facie case of obviousness by pointing to express or implicit suggestion to combine in the references themselves or by explaining or providing any basis for concluding that persons of skill in the art would be moved to combine the references. For these reasons also, Applicants continue to assert that the cited references are not properly combined in this case.

<u>The Cited References Set Forth No Suggestion Or</u>
<u>Motivation To Combine Nusbaum, Sun1, And Sun2</u>

The Final Office Action at page 3 again argues that the cited references set forth a suggestion or motivation to combine Nusbaum, Sun1, and Sun2. To establish a prima facie case of obviousness, there must be a suggestion or motivation to modify or combine Nusbaum, Sun1, and Sun2. *In re Vaeck*, 947 F.2d 488, 493, 20 USPQ2d 1438, 1442 (Fed. Cir. 1991). "The mere fact that references can be combined or modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination." *In re Mills*, 916 F.2d 680, 16 USPQ2d 1430 (Fed. Cir. 1990).

The Final Office Action does not point to any disclosure in Nusbaum, Sun1, or Sun2 suggesting the desirability of their combination or modification. Moreover, there is no possibility whatsoever that the Examiner could ever point to any disclosure in Nusbaum, Sun1, or Sun2 suggesting the desirability of the combination because none of the references teaches transcoding as claimed in the present application. In fact, Nusbaum makes no mention whatsoever of transcoding and makes no pertinent mention of email.

The Final Office Action however contends that Nusbaum teaches transcoding and cites Figure 8, page 18[1]. Nusbaum in Figure 8, however, in fact does not teach transcoding as claimed in the present application. Figure 8, on page 18, actually teaches a sample JSP (JavaServer Page) file and the Java code generated from the JSP file. Disclosing a sample JSP file and the Java code generated from a JSP file definitely does not teach transcoding as claimed in the present application.

The First Office Action at page 8 states that the title of Nusbaum, "Application Server," teaches a transcoding gateway. In fact, the title of Nusbaum, "Application Server," does not teach a transcoding gateway. The title of Nusbaum mentions 'application' and 'server.' An application is defined in Nusbaum at page 1 as a collection of user-supplied resources such as, for example, servlets, Enterprise Java beans, JavaServer Pages, static HTML, object groups and URLS. A 'server' edits, manages, deploys, runs and monitors applications. Disclosing the running and otherwise monitoring of applications, without more, is not a disclosure of a transcoding gateway.

As with Nusbuam, neither Sun1 nor Sun2 references transcoding as claimed in the present application. Because neither Nusbuam, Sun1, nor Sun2 teaches transcoding as claimed in the present application, none of the proposed references could possibly suggest the desirability of their combination for email administration as claimed in the present application. Absent such a showing of desirability, the Final Office Action has impermissibly used "hindsight" occasioned by applicants' own teaching to reject the

---

[1] The Final Office Action actually refers to page 8 rather than page 18. Figure 8, however, is set forth on page 18 of Nusbaum. Applicants therefore refer to page 18 rather than page 8.

claims 1-10, 13, 15-18, 23-32, 35, 37-40, 45-54, 57, 59-62. *In re Surko,* 11 F.3d 887, 42 U.S.P.Q.2d 1476 (Fed. Cir. 1997); *In re Vaeck,* 947 F.2d 488m 20 U.S.P.Q.2d 1438 (Fed. Cir. 1991); *In re Gorman,* 933 F.2d 982, 986, 18 U.S.P.Q.2d 1885, 1888 (Fed. Cir. 1991); *In re Bond,* 910 F.2d 831, 15 U.S.P.Q.2d 1566 (Fed. Cir. 1990); *In re Laskowski,* 871 F,.2d 115, 117, 10 U.S.P.Q.2d 1397, 1398 (Fed. Cir. 1989). Because the Final Office Action does not established a prima facie case of obviousness under 35 U.S.C. § 103(a), Applicants respectfully traverse each rejection individually of claims 1-10, 13, 15-18, 23-32, 35, 37-40, 45-54, 57, and 59-62 and request the claims be allowed.

There Is No Reasonable Expectation Of Success In The Combination Of Nusbaum, Sun1, And Sun2

In response to Applicants' Response to the First Office Action, the Final Office Action argues that there is a reasonable expectation of success in the combination of Nusbaum, Sun1, and Sun2. The Final Office Action does not however provide any new basis for this assertion other than to state:

> The claimed subject matter accomplishes a method, a system, a computer software product for email administration. Nusbaum discloses usage of a gateway/server for software administration (e.g., figure 8, page 8).

Nusbaum at Figure 8, actually on page 18 of Nusbaum, however, does not disclose usage of a gateway/server for software administration. Nusbaum at Figure 8 actually teaches a sample JSP (JavaServer Page) file and the Java code generated from the JSP file. Nusbaum's disclosure of Java Server Pages provides no basis whatsoever on which to found an expectation of success in combining Nusbaum with Sun1 and Sun2.

In fact, when viewed in perspective together, it is clear that neither Nusbaum, Sun1, nor Sun2 could ever possibly provide any basis for any expectation of success in combining Nusbaum, Sun1, and Sun2 to render obvious the claims of the present invention. Nusbaum generally teaches a kind of dynamic web page technology known as Java

Server Pages or 'JSP.' Nusbaum at section 1.4, page 13, states, for example, "JavaServer Pages (JSP) technology provides developers with an easy and powerful way to build Web pages with dynamic content." Sun1 at page 11 teaches a Java application programming interface ("API") that "provides a unified architecture and messaging protocol for managing the acquisition, processing, and delivery of time-based media data." Sun2 at page 1 teaches an API in Java that "provides a set of abstract classes defining objects that comprise a mail system." The dynamic web page technology of Nusbaum for building adhoc server pages is clearly not related to the Java API for time-based media or the Java API for a mail system of Sun2. In fact, the technology for building dynamic web pages is entirely different from a Java API for time-based media and a mail system. JSPs generate documents viewable in a web browser, while the APIs for time-based media and for mail systems provide a Java class architecture to software developers. Time-based media and email are not and cannot be implemented as part of dynamic web page technology without changing the principals of operation of the dynamic web page technology. There cannot possibly be to one of ordinary skill in the art at the time of the invention a reasonable expectation of success with the combination of Nusbaum, Sun1, and Sun2. The proposed combination of Nusbaum, Sun1, and Sun2 therefore cannot support a prima facie case of obviousness. The rejections to claims 1-66 should be withdrawn, and the case should be allowed.

Nusbaum Is Not Analogous Art Because It Is Neither In The Field Of Applicants' Endeavor Nor Reasonably Pertinent To The Particular Problem With Which The Applicants Were Concerned

In response to Applicants' First Office Action, the Final Office Action at pages 4 and 5 argues that Nusbaum is analogous art available for rejecting the claims of the present application. The Final Office Action asserts that Nusbaum is in the field of Applicants' endeavor or reasonably pertinent to the particular problem with which the Applicants were concerned. The only support for this assertion offered in the Final Office Action at page 5 states:

In this case, a system for email administration, as claimed, is similar to Nusbaum's teachings of computer device, i.e., gateway/server for software administration usage (e.g., figure 8, page 8), which is the same field of endeavor. The well-known software to support handling of email message contents that are taught by Sun2 would be supported by Nusbaum.

Nusbaum at Figure 8, on page 18, however teaches a sample JSP (JavaServer Page) file and the Java code generated from the JSP file. Without more, a sample JSP file and the Java code generated from the file does not place Nusbaum in the field of Applicants' endeavor or make Nusbaum reasonably pertinent to the particular problem with which the Applicants were concerned. In fact, Nusbaum cannot be a reference against the claims of the present application because Nusbaum does actually represents nonanalogous art within the meaning of *In Re Horn*, *Clay*, and *Oeitker*. *In re Horn*, 203 USPQ 969 (CCPA 1979), *In re Clay*, 966 F.2d 656, 23 USPQ2d 1058 (Fed. Cir. 1992), *In re Oeticker*, 977 F.2d 1443, 24 USPQ2d 1443 (Fed. Cir. 1992). The field of the inventors' effort in this case is routing email according to its content. The present application claims, among other things, receiving an email message in a transcoding gateway, transcoding into a digital file a digital object included in the email message, and downloading the digital file to a destination client device. The field of Nusbaum is dynamic web pages for the World Wide Web – which clearly has nothing to do with the technical field of the present application. Nusbaum therefore is not within the field of the inventor's endeavor in this case.

Because Nusbaum is not within the field of the inventor's endeavor in this case, there can be no basis for believing that Nusbaum as a reference would have been considered by one skilled in the particular art working on the relevant problem to which this invention pertains. That is, there would be no reason for an inventor concerned with transcoding gateways for email to search for art regarding dynamic generation of web pages. The two simply have nothing to do with one another. Nusbaum as a reference therefore is not reasonably pertinent to the particular problem with which the inventors were involved in the present case and is not available as a reference against the present application.

Applicants respectfully propose that for this reason alone the rejection of the present claims 1-66 should be withdrawn, and the claims should be allowed.

<div align="center">

Nusbaum, Sun1, and Sun2 Do Not Teach
Each and Every Element of the Claim

</div>

The Final Office Action at page 5 again argues that Nusbaum, Sun1, and Sun2 teach each and every element of independent claims 1, 23, and 45. To establish a prima facie case of obviousness, the proposed combination of Nusbaum, Sun1, and Sun2 must teach all of Applicants' claim limitations. *In re Royka*, 490F.2d 981, 985, 180 USPQ 580, 583 (CCPA 1974). In Applicants' Response to First Office Action, however, Applicants demonstrated that the combination of Nusbaum, Sun1, and Sun2 does not teach each and every element of independent claims 1, 23, and 45 because the limitations asserted to be disclosed by Sun2 in the First Office Action were, in fact, not taught by Sun2. The Final Office Action argues in response to Applicants' earlier demonstration that Sun2 does teach the limitations asserted in the First Office Action, citing new references in Sun2. Applicants propose that the new references to Sun2 in the Final Office Action however still do not teach the proposed limitations of claims 1, 23, and 45.

The Final Office Action at page 5, using a new reference from Sun2, states that Sun2 at page 6 teaches "receiving...an email message...." What page 6 of Sun2 actually teach is that "JavaMail clients use the JavaMail API and Service Providers implement the JavaMail API." Sun2's JavaMail client application that uses a JavaMail application programming interface implemented on a service provider is not "receiving...an email message" as claimed in the present application.

The Final Office Action at page 5, using another new reference from Sun2, states that Sun2 teaches "the email message having destination mailbox address and object...." Applicants however respectfully note that a limitation of "the email message having destination mailbox address and object..." is nowhere in claims 1, 23, or 45. Applicants

therefore need not respond as to whether this limitation is taught in Sun2 as asserted in the Final Office Action.

The Final Office Action at page 5, using a further new reference from Sun2, states that Sun2 at pages 54 and 57 teaches "a mailbox address field," "a mailbox address identical to the destination mailbox address of the email message," "an internet address field," and "a digital file format code field...." What pages 54 and 57 of Sun2 actually teach are a code segment that "sends a MimeMessage using a Transport class implementing the SMTP protocol" and a code sample that "creates a new MimeMessage object for sending." The sending a MimeMessage using a Transport class and creating a new MimeMessage object do not teach "a mailbox address field," "a mailbox address identical to the destination mailbox address of the email message," "an internet address field," and "a digital file format code field" as claimed in the present application.

The Final Office Action at page 5, using a further new reference from Sun2, states that Sun2 at pages 20 and 60 teach "a path name field...." What pages 20 and 60 of Sun2 actually teach is that "it is important to check the ContentType header for each BodyPart element stored within a Multipart container" because "Multipart objects can be nested to any reasonable depth within a multipart message" and that the "ContentType class is a utility class that parses and generates MIME content-type headers." The ContentType utility class that parses and generates MIME content-type headers and the checking ContentType headers for each BodyPart element stored in a Multipart container of Sun2 does not teach a path name field as claimed in the present application.

Even with the new references to Sun2 cited in the Final Office Action, the combination of Nusbaum, Sun1, and Sun2 still does not teach all the limitations of claims 1, 23, and 45. The Final Office Action therefore cannot establish a prima facie case for obviousness of claims 1, 23, and 45 under 35 U.S.C. § 103. The applicants respectfully propose that all the dependent claims in the present case stand because the independent claims 1, 23, and 45 stand. The rejection of all the claims 1 - 66 should therefore be withdrawn, and the claims should be allowed.
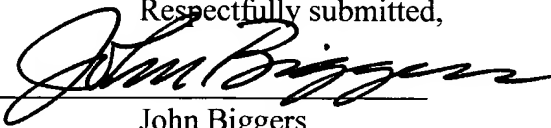
Conclusion

Applicants traverse each of the arguments in the Final Office Action responding to Applicants' Response to First Office Action. Applicants demonstrate that Nusbaum, Sun1, and Sun2 are not properly combined for an obviousness rejection under 35 U.S.C. § 103. Furthermore, the cited references in the Final Office Action set forth neither a suggestion nor motivation to combine Nusbaum, Sun1, and Sun2. In addition, there is no reasonable expectation of success in the combination of Nusbaum, Sun1, and Sun2. Moreover, Nusbaum represents nonanalogous art. Applicant also demonstrate that even with the new references to Sun2 cited in the Final Office Action, the combination of Nusbaum, Sun1, and Sun2 still does not teach all the limitations of claims 1, 23, and 45. Applicants' therefore respectfully traverse each rejection of claims 1-66 under 35 U.S.C. § 102(e).

In view of the forgoing arguments, reversal on all grounds of rejections is requested.

The Commissioner is hereby authorized to charge or credit Deposit Account No. 09-0447 for any fees required or overpaid.

Respectfully submitted,

Date: September 21, 2005   By:

John Biggers
Reg. No. 44,537
Biggers & Ohanian, LLP
P.O. Box 1469
Austin, Texas 78767-1469
Tel. (512) 472-9881
Fax (512) 472-9887
ATTORNEY FOR APPELLANTS

**APPENDIX OF CLAIMS
ON APPEAL IN PATENT APPLICATION OF
WILLIAM KRESS BODIN, *ET AL.*, SERIAL NO. 09/882,174**

CLAIMS

What is claimed is:

1.   A method of email administration,

the method implemented in a transcoding gateway, the transcoding gateway comprising client device records stored in computer memory, each client device record representing a client device, each client device record including

a mailbox address field,

an internet address field,

a digital file format code field, and

a path name field,

the transcoding gateway further comprising at least one file system, each file system further comprising file system storage locations, each file system storage location having a path name,

the method comprising the steps of:

receiving in the transcoding gateway an email message, the email message comprising at least one destination mailbox address, the email message further comprising at least one digital object;

transcoding the digital object into a digital file having a digital format and a file name; and

downloading the digital file to a destination client device at an internet address recorded in an internet address field of a client device record, the client device record having:

recorded in the client device record's mailbox address field, a mailbox address identical to the destination mailbox address of the email message, and,

recorded in the client device record's digital file format code field, a digital file format code indicating that the client device represented by the client device record is capable of receiving the digital format of the digital file.

2.   The method of claim 1 further comprising recording a multiplicity of client device records representing a multiplicity of client devices, including recording for each client device a mailbox address, an internet address where the client is to be found on an internet, a digital file format code identifying a digital file format that the client device is capable of receiving, and a path name identifying a location in a file system where digital files for each client device are to be stored.

3.   The method of claim 1 wherein receiving an email message further comprises receiving an email message by use of a standard email protocol.

4.   The method of claim 3 wherein the standard email protocol is SMTP.

5.   The method of claim 3 wherein the standard email protocol is POP3.

6.   The method of claim 1 wherein the file name includes a file name extension

identifying the digital format of the digital file.

7.   The method of claim 1 wherein each client device represented by a client device record comprises automated computing machinery, a web browser, and an internet client having an internet address.

8.   The method of claim 1 wherein the downloading is carried out by use of HTTP.

9.   The method of claim 1 further comprising the steps of:

storing the digital file in a file system location having a digital file path name identical to a path name recorded in a path name field in a client device record, the client device record having recorded in its mailbox address field a mailbox address equal to the mailbox address of the email message; and

encoding the digital file path name and the file name of the digital file into an HTML document having a conventional file name;

wherein downloading the digital file to the client device further comprises downloading the HTML document.

10.   The method of claim 9 wherein encoding the digital file path name and the file name of the digital file into an HTML document further comprises encoding a URL in a hyperlink in an HTML document.

11.   The method of claim 9 wherein the conventional file name is "index.html."

12.   The method of claim 9 wherein the conventional file name is "index.htm."

13.   The method of claim 9 further comprising storing the HTML document in the file

system location identified by the path name.

14.   The method of claim 9 wherein downloading the digital file further comprises:

receiving, from a client device, a first HTTP request message requesting the HTML file having the conventional file name, wherein the first HTTP request message includes a client internet address for the client device;

sending, in an HTTP response message to the client device, the HTML document having the conventional file name from a file system location identified a path name recorded in a client device record's path name field of a client record whose internet address field contains an internet address equal to the client internet address; and

receiving from the client device a second HTTP request message, wherein the second HTTP request message requests downloading of the digital file identified by the path name and the file name of the digital file encoded into the HTML document.

15.   The method of claim 1 wherein receiving an email message further comprises posting the email message to a destination mailbox at the destination mailbox address.

16.   The method of claim 1 further comprising delivering the email message from the destination mailbox to an email client, wherein the delivering is carried out by use of a standard email protocol.

17.   The method of claim 16 wherein the standard email protocol is POP.

18.   The method of claim 16 wherein the standard email protocol is POP3.

19. The method of claim 1 wherein the destination client device is an audio player and the digital format of the digital file is MP3.

20. The method of claim 1 wherein the destination client device is a video player and the digital format of the digital file is MPEG.

21. The method of claim 1 wherein the destination client device is a digital picture frame and the digital format of the digital file is JPEG.

22. The method of claim 1 wherein the destination client device is a digital picture frame and the digital format of the digital file is GIF.

23. A system for email administration,

the system implemented as a transcoding gateway, the transcoding gateway comprising client device records stored in computer memory, each client device record representing a client device, each client device record including

> a mailbox address field,
> an internet address field,
> a digital file format code field, and
> a path name field,

the transcoding gateway further comprising at least one file system, each file system further comprising file system storage locations, each file system storage location having a path name,

the system comprising:

means for receiving in the transcoding gateway an email message, the email message comprising at least one destination mailbox address, the email message further comprising at least one digital object;

means for transcoding the digital object into a digital file having a digital format and a file name; and

means for downloading the digital file to a destination client device at an internet address recorded in an internet address field of a client device record, the client device record having:

recorded in the client device record's mailbox address field, a mailbox address identical to the destination mailbox address of the email message, and,

recorded in the client device record's digital file format code field, a digital file format code indicating that the client device represented by the client device record is capable of receiving the digital format of the digital file.

24. The system of claim 23 further comprising means for recording a multiplicity of client device records representing a multiplicity of client devices, including means for recording for each client device a mailbox address, an internet address where the client is to be found on an internet, a digital file format code identifying a digital file format that the client device is capable of receiving, and a path name identifying a location in a file system where digital files for each client device are to be stored.

25. The system of claim 23 wherein means for receiving an email message further comprises means for receiving an email message by use of a standard email protocol.

27

26.   The system of claim 25 wherein the standard email protocol is SMTP.

27.   The system of claim 25 wherein the standard email protocol is POP3.

28.   The system of claim 23 wherein the file name includes a file name extension identifying the digital format of the digital file.

29.   The system of claim 23 wherein each client device represented by a client device record comprises automated computing machinery, a web browser, and an internet client having an internet address.

30.   The system of claim 23 wherein the means for downloading utilizes HTTP.

31.   The system of claim 23 further comprising:

    means for storing the digital file in a file system location having a digital file path name identical to a path name recorded in a path name field in a client device record, the client device record having recorded in its mailbox address field a mailbox address equal to the mailbox address of the email message; and

    means for encoding the digital file path name and the file name of the digital file into an HTML document having a conventional file name;

    wherein means for downloading the digital file to the client device further comprises means for downloading the HTML document.

32.   The system of claim 31 wherein means for encoding the digital file path name and the file name of the digital file into an HTML document further comprises means for encoding a URL in a hyperlink in an HTML document.

33. The system of claim 31 wherein the conventional file name is "index.html."

34. The system of claim 31 wherein the conventional file name is "index.htm."

35. The system of claim 31 further comprising means for storing the HTML document in the file system location identified by the path name.

36. The system of claim 31 wherein means for downloading the digital file further comprises:

> means for receiving, from a client device, a first HTTP request message requesting the HTML file having the conventional file name, wherein the first HTTP request message includes a client internet address for the client device;

> means for sending, in an HTTP response message to the client device, the HTML document having the conventional file name from a file system location identified a path name recorded in a client device record's path name field of a client record whose internet address field contains an internet address equal to the client internet address; and

> means for receiving from the client device a second HTTP request message, wherein the second HTTP request message requests downloading of the digital file identified by the path name and the file name of the digital file encoded into the HTML document.

37.   The system of claim 23 wherein means for receiving an email message further comprises means for posting the email message to a destination mailbox at the destination mailbox address.

38.   The system of claim 23 further comprising means for delivering the email message from the destination mailbox to an email client, wherein the means for delivering utilizes a standard email protocol.

39.   The system of claim 38 wherein the standard email protocol is POP.

40.   The system of claim 38 wherein the standard email protocol is POP3.

41.   The system of claim 23 wherein the destination client device is an audio player and the digital format of the digital file is MP3.

42.   The system of claim 23 wherein the destination client device is a video player and the digital format of the digital file is MPEG.

43.   The system of claim 23 wherein the destination client device is a digital picture frame and the digital format of the digital file is JPEG.

44.   The system of claim 23 wherein the destination client device is a digital picture frame and the digital format of the digital file is GIF.

45.   A computer software product for email administration,

the computer software product capable of implementation in conjunction with a transcoding gateway, the transcoding gateway comprising client device records stored in computer memory, each client device record representing a client device, each client device record including

a mailbox address field,

an internet address field,

a digital file format code field, and

a path name field,

the transcoding gateway further comprising at least one file system, each file system further comprising file system storage locations, each file system storage location having a path name,

the computer software product comprising:

a recording medium;

means, recorded on the recording medium, for receiving in the transcoding gateway an email message, the email message comprising at least one destination mailbox address, the email message further comprising at least one digital object;

means, recorded on the recording medium, for transcoding the digital object into a digital file having a digital format and a file name; and

means, recorded on the recording medium, for downloading the digital file to a destination client device at an internet address recorded in an internet address field of a client device record, the client device record having:

recorded in the client device record's mailbox address field, a mailbox address identical to the destination mailbox address of the email message, and,

recorded in the client device record's digital file format code field, a digital file format code indicating that the client device

represented by the client device record is capable of receiving the digital format of the digital file.

46. The computer software product of claim 45 further comprising means, recorded on the recording medium, for recording a multiplicity of client device records representing a multiplicity of client devices, including means for recording for each client device a mailbox address, an internet address where the client is to be found on an internet, a digital file format code identifying a digital file format that the client device is capable of receiving, and a path name identifying a location in a file system where digital files for each client device are to be stored.

47. The computer software product of claim 45 wherein means for receiving an email message further comprises means, recorded on the recording medium, for receiving an email message by use of a standard email protocol.

48. The computer software product of claim 47 wherein the standard email protocol is SMTP.

49. The computer software product of claim 47 wherein the standard email protocol is POP3.

50. The computer software product of claim 45 wherein the file name includes a file name extension identifying the digital format of the digital file.

51. The computer software product of claim 45 wherein each client device represented by a client device record comprises automated computing machinery, a web browser, and an internet client having an internet address.

52. The computer software product of claim 45 wherein the means for downloading utilizes HTTP.

53. The computer software product of claim 45 further comprising:

> means, recorded on the recording medium, for storing the digital file in a file system location having a digital file path name identical to a path name recorded in a path name field in a client device record, the client device record having recorded in its mailbox address field a mailbox address equal to the mailbox address of the email message; and

> means, recorded on the recording medium, for encoding the digital file path name and the file name of the digital file into an HTML document having a conventional file name;

> wherein means for downloading the digital file to the client device further comprises means, recorded on the recording medium, for downloading the HTML document.

54. The computer software product of claim 53 wherein means for encoding the digital file path name and the file name of the digital file into an HTML document further comprises means, recorded on the recording medium, for encoding a URL in a hyperlink in an HTML document.

55. The computer software product of claim 53 wherein the conventional file name is "index.html."

56. The computer software product of claim 53 wherein the conventional file name is "index.htm."

57. The computer software product of claim 53 further comprising means, recorded on the recording medium, for storing the HTML document in the file system location identified by the path name.

58. The computer software product of claim 53 wherein means for downloading the digital file further comprises:

> means, recorded on the recording medium, for receiving, from a client device, a first HTTP request message requesting the HTML file having the conventional file name, wherein the first HTTP request message includes a client internet address for the client device;

> means, recorded on the recording medium, for sending, in an HTTP response message to the client device, the HTML document having the conventional file name from a file system location identified a path name recorded in a client device record's path name field of a client record whose internet address field contains an internet address equal to the client internet address; and

> means, recorded on the recording medium, for receiving from the client device a second HTTP request message, wherein the second HTTP request message requests downloading of the digital file identified by the path name and the file name of the digital file encoded into the HTML document.

59. The computer software product of claim 45 wherein means for receiving an email message further comprises means, recorded on the recording medium, posting the email message to a destination mailbox at the destination mailbox address.

60. The computer software product of claim 45 further comprising means, recorded on the recording medium, for delivering the email message from the destination mailbox to an email client, wherein the means for delivering utilizes a standard email protocol.

61.   The computer software product of claim 60 wherein the standard email protocol is POP.

62.   The computer software product of claim 60 wherein the standard email protocol is POP3.

63.   The computer software product of claim 45 wherein the destination client device is an audio player and the digital format of the digital file is MP3.

64.   The computer software product of claim 45 wherein the destination client device is a video player and the digital format of the digital file is MPEG.

65.   The computer software product of claim 45 wherein the destination client device is a digital picture frame and the digital format of the digital file is JPEG.

66.   The computer software product of claim 45 wherein the destination client device is a digital picture frame and the digital format of the digital file is GIF.

Application Server Solution Guide
Enterprise Edition: Getting Started

Barry Nusbaum, Bill Moore, Cristian E. Roldan
David Turner, Matthew Alcock, Steen Colliander
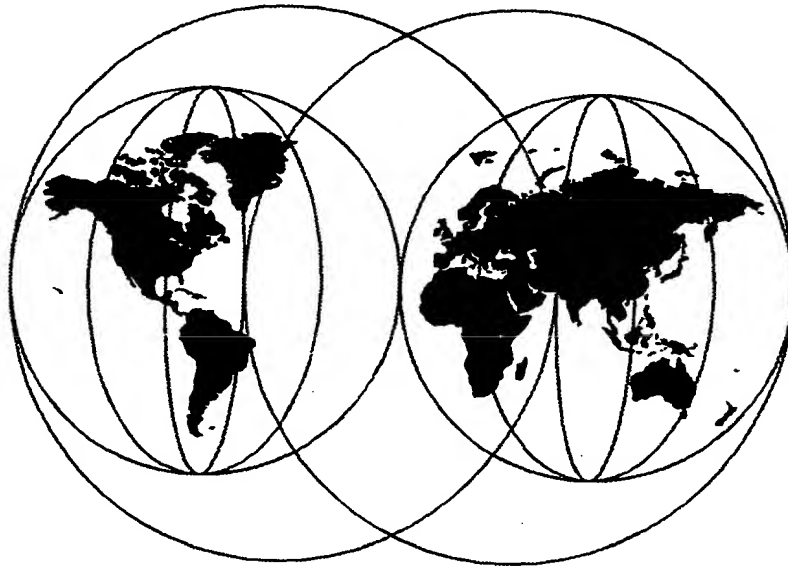
IBM International Technical Support Organization

Application Server Solution Guide
Enterprise Edition: Getting Started

May 2000

> **Take Note!**
>
> Before using this information and the product it supports, be sure to read the general information in Appendix C, "Special notices" on page 557.

First Edition (May 2000)

This edition applies to Versions 3.0 and 3.02 of WebSphere Application Server Enterprise Edition for use with Windows NT and AIX.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 678
P.O. Box 12195
Research Triangle Park, NC 27709-2195

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Preface

This redbook will help you install, configure, and use WebSphere Application Server Enterprise Edition. We take an early look at the tools and technologies included in the Enterprise Edition package, with our focus on the Enterprise Java programming model, as opposed to the CORBA model that is also supported by the Enterprise Edition. The redbook, *WebSphere Application Server Enterprise Edition Component Broker 3.0 First Steps*, SG24-2033, discusses the use of the CORBA model.

This redbook gives a broad understanding of Enterprise Edition architecture and an introduction to the setup and use of some of its key components. It will be particularly useful to first-time users of Enterprise Edition. We assume that you have a good understanding of the Java language and some knowledge of Web-based application development.

Chapter 1, "WebSphere applications" on page 1 contains an introduction to WebSphere applications, including an overview of the key components of the Enterprise Java programming model, and what tools are used to edit, manage, deploy, run, and monitor applications.

Chapter 2, "WebSphere Application Server overview" on page 31 is a high-level overview of the architecture and functions of the Enterprise Java Server deployed in IBM WebSphere. There are also details on installation. More detail is given in Chapter 3, "Managing the infrastructure" on page 47 on how to manage servlets, Web applications, JSPs, servlet engines and EJBs in this environment.

Chapter 4, "Development tools" on page 87 is about how to use IBM-supplied development and test tools to build Java applications for deployment in WebSphere.

Chapter 5, "EJB deployment" on page 295 is a more detailed look at deployment of EJBs into WebSphere.

Chapter 6, "WebSphere interoperability" on page 337 considers how Enterprise Java applications in WebSphere can interact and operate with other Enterprise products that are part of the entire WebSphere Enterprise Edition. It includes information on interaction with TXSeries and Component Broker.

Chapter 7, "WebSphere access control and security" on page 363 looks at the WebSphere security model and how it interacts with other products including DCE, SecureWay Policy Director, and RACF. This is expanded

further in Chapter 8, "Directory Services" on page 403 where we look at Directory Services and how to configure WebSphere to use LDAP.

Chapter 9, "Debugging logging and tracing" on page 449 considers how to solve problems with the facilities provided by WebSphere, including debugging tools.

Finally in Chapter 10, "WebSphere 3.0 samples" on page 493 we look at the standard samples distributed with WebSphere and describe how to configure and run these samples.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

Barry Nusbaum is a Senior Technical Consultant at the Solution Partnership Center in Waltham, Massachusetts. He writes extensively and teaches IBM classes worldwide on all areas of UNIX and Windows systems management. He is also currently working on projects related to the UNIX operating system. Before joining the Solution Partnership Center, he worked in the ITSO for seven years. He can be reached at bnusbaum@us.ibm.com.

Bill Moore is a Senior Aim Consultant at the IBM Transarc lab in Sydney, Australia. He has 15 years of application development experience on a wide range of computing platforms and using many different coding languages. He holds a Master of Arts degree in English from the University of Waikato, in Hamilton, New Zealand. His current areas of expertise include the VisualAge family of application development tools, object-oriented programming and design, and e-business application development. He can be reached at billm@au1.ibm.com.

Cristian E. Roldan is a WebSphere Specialist in IBM Argentina. He has four years of experience in AIX and three years in Java and other development tools for e-business. He is also currently working on projects related to e-business and WebSphere for several platforms. He holds a degree from the University of Salvador at Buenos Aires, Argentina. He has worked at IBM for six years. His areas of expertise include WebSphere Application Server, VisualAge Java and AIX. He can be reached at roldanc@ar.ibm.com.

David Turner is a Consulting I/T Architect for the IBM Transarc Lab in Pittsburgh and is based in Austin, Texas. He has 20 years of experience in a wide variety of computing environments, including all IBM platforms and tools. He holds a Bachelors of Arts degree from the University of Texas at

Austin and is a certified TXSeries CICS monitor specialist. His areas of expertise include project planning and management, CICS, IMS, DB2, and distributed system design and deployment. You can reach him at dtturner@us.ibm.com.

Matthew Alcock is an IT Specialist at IBM Global Services Australia. He has over two years of experience in working with IBM software technologies. He holds a Bachelor of Science degree in Computing Science from the University of Technology, Sydney. His areas of expertise include AIX, Windows NT, DB2 UDB and WebSphere.

Steen Colliander is a Senior IT Specialist in IBM Denmark. He has worked at IBM for 10 years. His areas of expertise include AIX and solutions within the e-business Application Framework with a focus on the IBM WebSphere product family and IBM Net.Commerce.

Thanks to the following IBM employees:

Nataraj Nagaratnam, WebSphere Security, Raleigh

Thomas Alcott, WebSphere Technical Sales Support, Costa Mesa

Ashok Iyengar, Customer Solutions Development, Pittsburgh

Melissa Pike, Customer Solutions Development, Pittsburgh

Inga Chapman, CICS Development, Hursley

Jenny Walls, Hursley

Wayne Winsor, CICS Support, Raleigh

Maggie Archibald, Performance Tuning, Dallas

Heinz Johner, DCE and Security, Austin

Larry Schick, Security Specialist, Los Angeles

Morgan Kinne, WebSphere Studio Development, Raleigh

Vincent Illiano, WebSphere Application Server Development, Raleigh

Sajith Kizhakkiniyil, WebSphere Studio Development, Mountain View

Chapter 1. WebSphere applications

This chapter provides an overview of WebSphere applications. Throughout this chapter we take a look at what an application is, what components are used to form applications, and what is used to edit, manage, deploy, run and monitor applications.

1.1 What is an application?

An application in the context of WebSphere is a collection of user-supplied resources. Some examples of this are servlets, Enterprise Java beans (EJB), JavaServer Pages (JSP), static HTML, object groups and URLs.

An application is created so that a group of resources can be managed as a single entity, enabling users to easily specify conditions such as startup and exit dependencies over the entire group rather than over each component.

WebSphere's description of an application is an object that represents a collection of *live objects* with specific startup dependencies defined by the user. Applications are *top-level* objects, and as such, they cannot be contained by any other repository object.

The following section provides a brief overview of the major components used to make up an application. For further information refer to Chapter 3, "Managing the infrastructure" on page 47.

1.2 What is a servlet?

Servlets are small, server-side, platform-independent Java programs that enable a Web server to extend its capabilities with minimal overhead, maintenance and support. Servlets are compiled bytecode which can be dynamically loaded, or downloaded across a network, with no platform-specific considerations or modifications. This provides servlets with the capability of being written once and run anywhere.

Servlets are analogous to Java applets in the sense that the applets run on the client machine and the servlet, being a Java program like an applet, runs on the server. Just the fact that the servlets run on the server is a big advantage over applets because they have access to the server's resources. Servlets can then exercise a lot more control on what information should be sent out to the clients. In addition, there is no requirement for servlets to be graphically visible to the user, whereas, applets would be.

Servlets use a standard Java Application Programming Interface (API) as described in 1.2.2, "Java Servlet API V2.1 overview" on page 4 along with the associated classes and methods, which are supported by most major Web servers. Servlets can also use additional Java classes and packages to extend the Java Servlet API 2.1.

Servlets communicate through requests and responses, similar to the behavior of HyperText Transfer Protocol (HTTP). They interact with a servlet engine running on a Web server by using these requests and responses. For example, when a client sends a request to the server, the server can send the request information onto the servlet and have the servlet process the request, form a response which the server can then send back to the client.
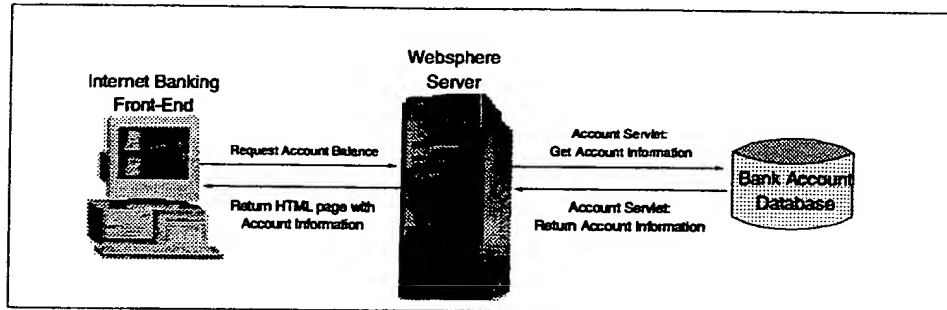
Figure 1. Example of servlet communication and interaction

Servlets can be loaded automatically when an applications is loaded, or they can be loaded the first time a client requests its services. Upon being loaded, a servlet will continue to run waiting for additional client requests. In addition, by using servlet aliases (servlet URLs), multiple instances are able to be created.

After the code for the servlet has been written, it has to be compiled, just like any other Java source code to generate the corresponding class file. To compile the servlet code, any Java tool/IDE can be used. After the code compiles without any errors and creates a class file, then we have to copy these files into appropriate directories. Other than the Java source code and the class file, we may have other associated JSP and html files. All these files need to go in the right folders before the servlet can be invoked.

1.2.1 Servlet lifecycle

Each servlet has the same lifecycle. It begins with the server engine loading the servlet into its memory and initializing it. The servlet is then ready to

handle requests from zero or more clients, until when the server removes the servlet.

1.2.1.1 Initializing a servlet

After the servlet is loaded into memory, the servlet engine attempts to create an instance of the servlet. This is usually done at the application startup, if that option is activated for the servlet, or at the first client request for the servlet after the application's startup.

The servlet engine creates an instance by creating the servlet configuration object and uses it to pass the servlet's initialization parameters to the *init* method. These parameters are applied to all invocations of the servlet until it is destroyed. The server may only load the servlet once. It must first destroy the servlet before reloading a new instance of the servlet.

If the initialization completes successfully, then the servlet is available to handle client requests. However, if the initialization fails then the servlet engine unloads the servlet. It is also possible that the administrator can set an application and its servlets to be unavailable for service. If this is the case the application and servlets will be unable to be run until the administrator changes their status to available.

1.2.1.2 Handling client requests

When a client request is received by the application server, the servlet engine creates a request object and a response object. These objects are then passed to the servlet *service* method when invoked by the servlet engine.

The service method gets information about the request from the request object. It then processes the request. It does this by invoking other methods such as doGet, doPost, doPut, doDelete or your own methods. It then uses the methods of the response object to pass back the response to the client.

1.2.1.3 Destroying a servlet

Servlets run until the servlet engine invokes the servlet's destroy method. This is usually caused by a request from the system administrator for an application to be stopped, causing the servlets belonging to that application to also be stopped.

Once the servlet is stopped the server then unloads the servlet. The Java Virtual Machine (JVM) performs a garbage collection sometime after the destroy.

1.2.2 Java Servlet API V2.1 overview

The Java Servlet Application Programming Interface (API) has been designed for today's and future request-response protocols. This expendability is provided by the API comprising two packages:

1. An HTTP-specific package

2. A non-HTTP-specific package

The IBM WebSphere Application Server Version 3.0 provides a servlet engine that implements the Java Servlet API V2.1. The application server implements the Java Servlet API V2.1 by including its packages in the application servers servlet.jar. The packages are:

- javax.servlet

- javax.servlet.http

These two packages contain seven interfaces, five classes and two exceptions. More information on these interfaces, classes and exceptions and the structure of a servlet can be found in the redbook *WebSphere Studio and VisualAge for Java - Servlet and JSP Programming*, SG24-5755-00. Also the JavaDoc documentation for the Java Servlet API V2.1 can be found at: http://java.sun.com/products/servlet/2.1/api/packages.html.

1.2.3 IBM extensions to the Servlet API V2.1

In its attempt to make it easier to manage session states and to create personalized Web pages, the IBM WebSphere Application Server Version 3.0 has included its own packages that extend and add to the Java Servlet API V2.1. The additional packages and classes are:

- com.ibm.websphere.servlet.personalization.sessiontracking package

- com.ibm.websphere.servlet.personalization.userprofile package

- com.ibm.websphere.db package

- com.ibm.websphere.servlet.error.ServletErrorReport class

- com.ibm.websphere.servlet.event package

- com.ibm.websphere.servlet.filter package

- com.ibm.websphere.servlet.request package

- com.ibm.websphere.servlet.response package

1.2.4 Servlet API V2.1 details

This section describes the functions of the Java Servlet API V2.1 as well as the IBM extensions to the API.

As listed in 1.2.3, "IBM extensions to the Servlet API V2.1" on page 4, the following packages and classes were added to extend the Java Servlet API V2.1 to make it easier to manage session state and to create personalized Web pages, and below are some details on what they actually provide.

com.ibm.websphere.servlet.personalization.sessiontracking package

This package provides the following:

- Records the referral page that led the visitor to your Web site.
- Tracks the visitor's position within the site.
- Associates user identification with the session.

com.ibm.websphere.servlet.personalization.userprofile package

This package provides the following:

- An interface for maintaining detailed information about visitors to your Web site. This is done by storing the information in a database.
- The ability to create Web applications that incorporate the detailed information, allowing you to create a personalized experience for each of your visitors.

com.ibm.websphere.db package

This package provides the following:

- Simplifies access to relational databases.
- Enhanced access functions, for example result caching, update through the cache, and query parameter support.

com.ibm.websphere.servlet.error.ServletErrorReport class

This class enables the application to provide more detailed and tailored messages when errors occur.

com.ibm.websphere.servlet.event package

This package provides the following:

- Listener interfaces for notifications of lifecycle events for applications and servers as well as servlet errors.

• An interface for registering listeners.

com.ibm.websphere.servlet.filter package

This package provides the following:

• Support for servlet chaining.

• The *ChainerServlet*, which needs to be added to an application to give it the ability to chain servlets.

• The *ServletChain* object.

• The *ChainResponse* object.

com.ibm.websphere.servlet.request package

This package provides the following:

• The *HttpServletRequestProxy* abstract class is used to overload the servlet engine's *HttpServletRequest* object, causing the overloaded request object to be forwarded to another servlet for processing.

• The *ServletInputStreamAdapter* class is used to convert an *InputStream* into a *ServletInputStream* and proxying all method calls to the underlying *InputStream*.

com.ibm.websphere.servlet.response package

This package provides the following:

• The *HttpServletResponseProxy* abstract class is used to overload the servlet engine's *HttpServletResponse* object, causing the overloaded response object to be forwarded to another servlet for processing.

• The *ServletOutputStreamAdapter* class is used to convert an *OutputStream* into a *ServletOutputStream* and proxying all method calls to the underlying *OutputStream*.

• The *StoredResponse* object is used to cache a servlet's response that contains data that is not expected to change for a period of time.

1.2.5 Changes to packages supported in WAS V2

The *com.ibm.servlet.connmgr* was used in the Application Server Version 2 to let a servlet communicate with a connection manager, that maintained a pool of open data server connections to JDBC or ODBC server products. This allowed the servlet to communicate directly with the data server using the connection manager's APIs when a connection was made from the pool.

This package has been deprecated, as the Application Server Version 3 now has a built-in connection pooling function. This now allows developers to write servlets to use the JDBC APIs to access the connection pool instead of using the connection manager's APIs directly.

Also in the Application Server Version 3 the following packages have been removed:

- *com.ibm.servlet.personalization.sam*
- *com.ibm.servlet.servlets.personalization.util*

1.3 What are Enterprise Java beans (EJB)?

This section will describe the Enterprise Java beans (EJBs) in the context of WebSphere V3.0. This section doesn't provide a detailed description of how to write EJBs. For that, the reader is encouraged to refer to other documents on that topic. A good source of reference on how to write EJBs in WebSphere is the IBM book *Writing Enterprise Beans in WebSphere*, SC09-4431-01. This book is shipped as part of the documentation with WebSphere V3.0. It can also be viewed or downloaded from:

http://www.ibm.com/software/webservers/appserv/library.html

1.3.1 Introduction

The EJBs are based on the component architecture. They are Java components which are used in a distributed client server environment. Typically, EJBs reside in the middle layer or the application layer (where the business logic resides) and provide the business logic implementation. These EJB components are lightweight, modular and easy to deploy. EJBs can be combined with other components to build an enterprise application. IBM's implementation of EJBs is based on the Sun Microsystems Enterprise JavaBeans specifications which can be found at:
http://java.sun.com/products/ejb/docs.html

There are two main categories of enterprise beans. They are shown in Figure 2 on page 8.
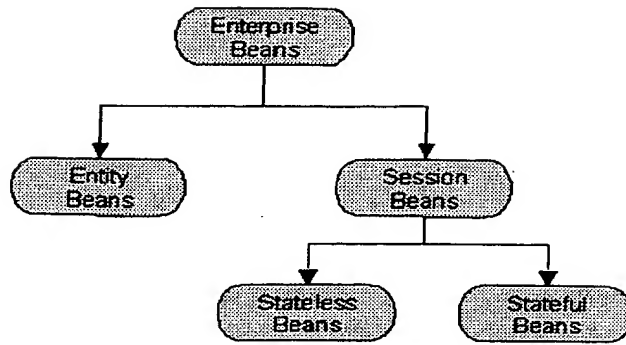
Figure 2. Categories of enterprise beans

1.3.1.1 Entity beans

Entity beans encapsulate permanent data and associated methods to manipulate the data. For their data to be permanent it needs to be stored somewhere, usually within a file system or a database such as IBM UDB DB2 or any other supported relational or object-oriented database. Instances of an entity bean are unique, but each bean can be accessed by multiple users.

The data can be synchronized in two ways. When the bean handles its own data synchronization, the process is called bean managed persistence (BMP). On the other hand when the container handles the data synchronization, the process is called container managed persistence (CMP). The entity bean lifecycle is shown in Figure 3 on page 9.
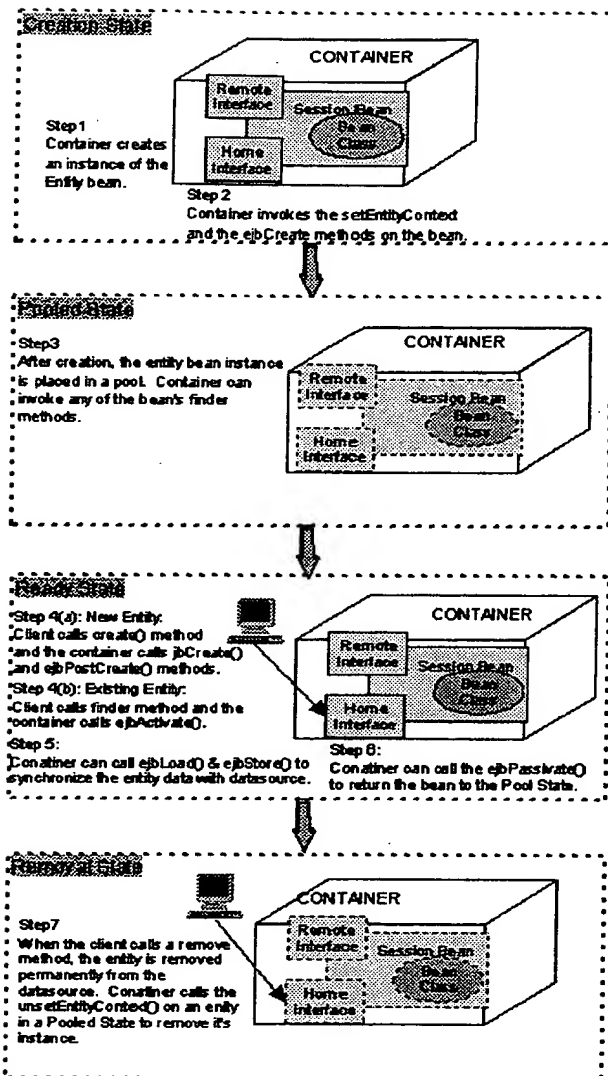
Figure 3. Entity bean life cycle

Each entity bean consists of the following 4 components:

a. Bean Class: This class is implemented by the developer to encapsulate the business methods and data. This class is hidden from the clients

and is accessed by the remote interfaces implemented by the container.

b. Home Interface: Container implements this interface and provides methods to create, find and remove the instances of the enterprise bean.

c. Remote Interface: This is the other interface that is implemented by the container when the enterprise bean is deployed in a container. The remote interface provides the clients access to the business data and methods in the bean class.

d. Primary Key Class: Since entity bean instances are unique, this class encapsulates one or more variables and methods to manipulate those variables, which are used as a primary key to uniquely identify a bean instance.

1.3.1.2 Session beans

Session beans encapsulate non-permanent data. They perform units of work on behalf of an EJB client. Typically, the session beans lifetime is that of the EJB client it is servicing. Unlike the entity beans, their data is not required to be stored in a data source. Session beans can however, make this data persistent by using underlying entity beans.

Every session bean has the following three components:

- Bean class
- Home interface
- Remote interface

The attributes of these interfaces and classes are the same as those of the entity beans. The reader will observe that the session bean does not have a Primary Key class as does the entity bean. The reason for this is that unlike the unique entity bean instances, the session bean instances are not unique. These instances are identical. When session beans encapsulate any semi-permanent data, then their instances become unique.

Each session bean is associated with one client. A session bean lifecycle is outlined in Figure 4 on page 11. Depending on the life span of a session bean they can be further classified as Stateful or Stateless session beans. A Stateful session bean maintains data across methods and thus has a longer life-span. On the other hand a Stateless session bean does not maintain data across the methods and is short-lived.
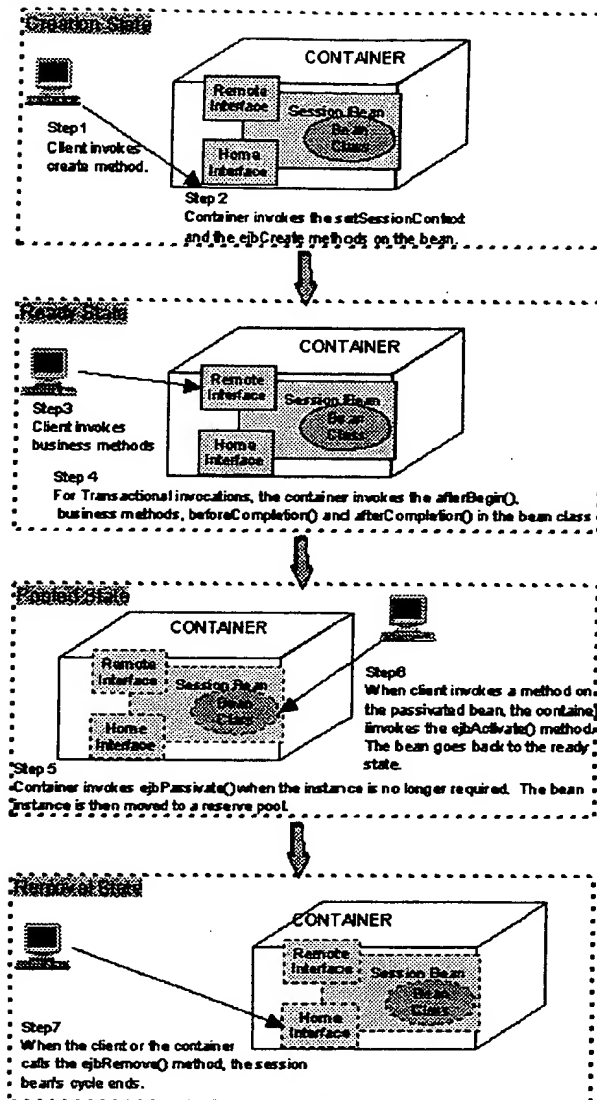
Figure 4. Session bean life cycle

1.3.2 EJB architecture in brief

The EJB architecture is based on the Sun Microsystems Enterprise JavaBeans specification. The EJB environment allows the users of the WebSphere Advanced and the WebSphere Enterprise editions to integrate their Web-based systems with their other business systems.

The EJB implementation consists of four major components. These are shown in Figure 5 on page 13.

1. EJB server

   In a 3-tier architecture scheme, the EJB server is the application server layer. As shown in Figure 5 on page 13, the EJB server connects the clients (for example servlets, JSPs or Java applications) to the enterprise data. The EJB server contains the business logic in the form of Java beans. These Java beans are deployed in a container, and a container is deployed in the EJB server. The clients cannot access the business logic and the business data in the Java beans directly. They have to go through the remote interface implemented by the container. The Advanced edition of WebSphere ships with only one EJB server called the EJB server (AE). The Enterprise edition of WebSphere ships with two EJB servers namely - the EJB server (AE) and the EJB server (CB). The EJB server (AE) includes only one standard container that supports both the session and the entity beans. The EJB server (CB) has two containers - the entity container for the entity beans, and the session container to hold the session beans.

   The reader can find detailed information on EJB servers in *Writing Enterprise Beans in WebSphere*, SC09-4431-01, that ships with WebSphere V3.

   This book can also be viewed or downloaded from:

   http://www.ibm.com/software/webservers/appserv/library.html

2. Data source

   The persistent data in the entity beans is stored in a recoverable data source. For container managed persistence (CMP), the EJB server (AE) supports IBM DB2, Oracle, and Microsoft SQL Server, and the EJB server (CB) supports DB2, Oracle, IBM CICS and IBM IMS.

   For bean managed persistence (BMP), the user can use any data source or a file stem to store the persistent data. The user will then have to write code for the beans to handle their own data source interactions.

3. EJB clients

The EJB clients can be one or more of the following: Java servlet, Java applet-servlet combination, or a JSP file. A Java applet can be used with a servlet to interact with the enterprise beans, while in the EJB server (CB), a Java applet can directly interact with the Enterprise Java beans. In an EJB server (CB) environment, additional EJB clients can be ActiveX clients, a CORBA-based Java clients, and to a limited degree a C++ CORBA clients. More details on how to write EJB clients can be obtained by the reader from *Writing Enterprise Beans in WebSphere*, SC09-4431-01.

4. Administration interface

The EJB server (AE) uses the WebSphere Administrative Console to administer the EJB server. The EJB server (CB) uses the System Management End User Interface (SMEUI).
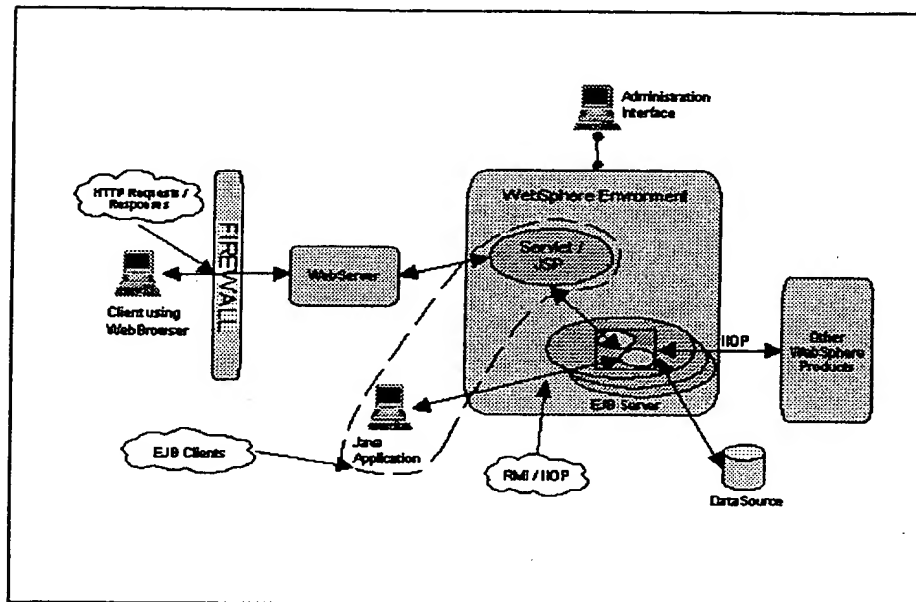
Figure 5. EJB environment and interaction with other components

1.4 What are JavaServer Pages (JSP)

JavaServer Pages (JSP) technology provides developers with an easy and powerful way to build Web pages with dynamic content. JSPs dynamically generate HTML, eXtensible Markup Language (XML), and other structured

documents inside a server, and enable you to effectively separate the structured documents from the business logic in your Web pages.

The IBM WebSphere Application Server programming model implements the JavaSoft JSP Specification. In addition IBM has added to the JSP specifications with JSP tags that are HTML-like which will make it easier for HTML authors to add the power of Java to their Web pages. The Application server provides support for two levels of the JSP specifications, which are JSP 1.0 and JSP 0.91.

JSP technology-enabled pages share the same ability as servlets, in that they are written once, but can be run anywhere.

1.4.1 JSP process flow

The JSP process flow is shown in Figure 6 on page 15.

The JSP processors puts the .java and the .class files inside the \WebSphere\AppServer\Temp folder. The location of these files will depend on whether you are using JSP 0.91 or JSP 1.0 in your application.

For example, if you are using JSP 0.91 and the JSP file is in the following folder:

<WASRoot>\hosts\default_host\examples\web, then the JSP processor will place the .java and the .class files in the following path:

<WASRoot>\temp\examples\pagecompile folder.

The JSP 0.91 processor uses a naming convention to name these .java and the .class files. If, for example, the JSP filename is simple.jsp, then the processor will name the .java and the .class files as _simple_xjsp.java and _simple_xjsp.class, respectively. Under JSP 0.91 the .java file is always kept.

If you are using JSP 1.0 and the JSP file is in the following folder:

<WASRoot>\hosts\default_host\examples\web, then the JSP processor will place the .class files in the following path:

<WASRoot>\temp\examples

The JSP 1.0 processor names the .class file simple.class and by default the .java file is not kept after compilation. To keep the .java file it is necessary to set the Init Parm Name keepgenerated used by the JspServlet to true. You can do this from the WebSphere Administrative console as shown in Figure 7 on page 17
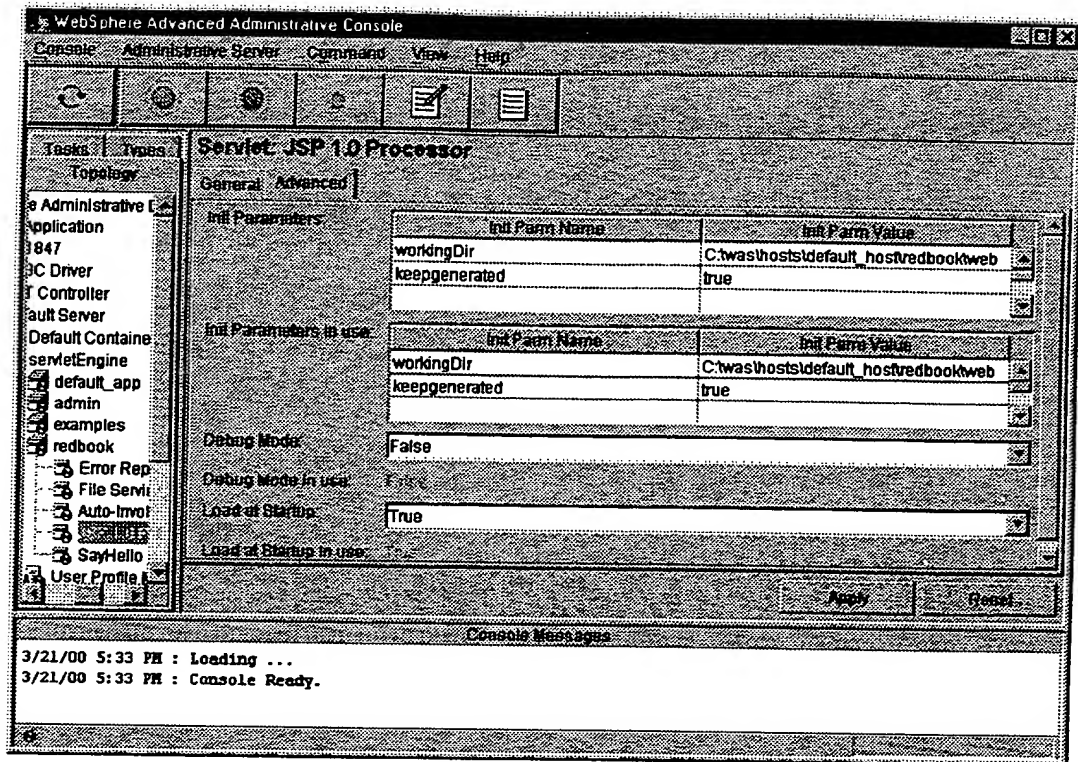
Figure 7. Setting Init Parm Name for JspServlet

The .java and the .class files generated from the JSP file are servlets extending the javax.servlet.http.HttpServlet class. A sample JSP 0.91 file and the Java code generated by it is shown in Figure 8 on page 18.

```
<BEAN name="simpleSessionMsg" type="SimpleJSPBean" create="true"
scope="session"></BEAN>
<BEAN name="simpleRequestMsg" type="SimpleJSPBean" create="true"
scope="request"></BEAN>
<%
    SimpleJSPBean simpleApplicationMsg =
    (SimpleJSPBean)
getServletContext().getAttribute("simpleApplicationMsg");
%>


<html>
<head><title>Simple JSP</title></head>
<body>
```

```
<img src="banner.gif" alt="banner image" height=100 width=584><p>
<h1>Simple JSP page</h1>
<% if(simpleApplicationMsg != null){ %>
<B>Application Bean:</B>  <%=simpleApplicationMsg.getMessage() %><BR>
<% } %>


<B>Session Bean:</B> <%= simpleSessionMsg.getMessage() %> <BR>


<B>Request Bean:</B> <%= simpleRequestMsg.getMessage() %> <BR>

</body>
</html>
```

Figure 8. simple.JSP code 0.91

When this file is processed by the JSP processor, the _simple_xjsp.java file is generated. In Figure 9 on page 19, only a snippet of code is shown. Please refer to Appendix B, "Sample Code" on page 549, for all of the code and an example of the code generated by the JSP 1.0 compiler.

```
package pagecompile;


import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.beans.Beans;
import com.ibm.servlet.jsp.http.pagecompile.ParamsHttpServletRequest;
import com.ibm.servlet.jsp.http.pagecompile.ServletUtil;
import com.ibm.servlet.jsp.http.pagecompile.filecache.CharFileData;
import com.ibm.servlet.jsp.http.pagecompile.NCSAUtil;


import SimpleJSPBean;

public class _simple_xjsp extends javax.servlet.http.HttpServlet {
    private static final String sources[] = new String[] {
        "c:\\websphere\\appserver\\hosts\\default_host\\examples\\
web\\simple.jsp",
    };
    private static final long lastModified[] = {
        926708647000L,
    };

    public void service(HttpServletRequest request,
HttpServletResponse response)
```

```
                    throws IOException, ServletException
        {
    --------------
    --------------
        }


            SimpleJSPBean simpleSessionMsg= (SimpleJSPBean)
                request.getAttribute("simpleSessionMsg");
            if ( simpleSessionMsg == null )
                throw new ServletException("Invalid BEAN name:
simpleSessionMsg");
            {
            java.util.Properties p = new java.util.Properties();
            java.util.Enumeration e = request.getParameterNames();
            while (e.hasMoreElements()) {
                String name = (String) e.nextElement();
                p.put(name, request.getParameter(name));
            }
            com.ibm.servlet.util.BeansUtil.setProperties(simpleSessionMsg, p);
            }
    --------------------
    --------------------
        }
```

Figure 9. Code snippet of the _simple_xjsp.java file generated by the JSP processor

1.4.2 JSP lifecycle

Since after compilation, JSPs generate a servlet, their life cycle is similar to that of a servlet. When the ServletEngine receives a request for a JSP file, it checks to see if the servlet already exists or if the JSP file has changed since the last time it was invoked. If the servlet for the JSP does not exist in the application classpath or if the JSP file was changed since the last time it was loaded, the servlet engine passes the request to the JSP processor (or the pagecompile servlet). This creates another .java and .class file for the requested JSP file. These files are placed in the application classpath. The servletengine then creates an instance of the class file and calls the servlet service() method in response to the request. Once the .class and .java files have been created by the JSP processor, all the subsequent requests for the JSP servlet are handled by the instance of the servlet that was created by the servletengine. By default, the JSP syntax in a JSP file is converted to Java code by the processor and this code is placed in the service() method of the generated class file. This default behavior can be overridden by using the method directive in the JSP file.

The JSP servlet is terminated when the servletengine no longer needs the servlet or a new instance of the servlet is being loaded by the servletengine.

In doing so, the destroy() method in the servlet is called. If there is a need to conserve resources or if the previous request to the servlet times out, then also, the servletengine can call the destroy() method of the servlet.

1.4.3 JSP access models

When using JSPs there are commonly two types of access models used:

1. JSP file handles both the request and the response

   In this model the JSP file handles both the request and the response to and from the client browser. The JSP passes the request to beans or other components to generate the dynamic content. The response is sent back to the JSP file from the beans, which in turn sends it back to the client browser.

2. JSP handles response, servlet handles request

   In this model, the client request is sent to the servlet, which handles the dynamic content generation. It then calls a JSP file to send the response back to the client. Using this model, one can effectively separate the business logic from the content display.

These two JSP access models are discussed in more detail in the redbook *Patterns for e-business:User to Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition, SG24-5864-00.*

1.4.4 JSP syntax

In WebSphere V3, JSP conforms to the JavaServer Pages Specifications 0.91 or 1.0 from Sun Microsystems. IBM has added some enhancements particularly in the area of database access. The Sun JSP 1.0 specification can be found at:

        http://java.sun.com/products/jsp/download.html

A full discussion of JSP syntax including the differences between Version 0.91 and 1.0, and details of the IBM extensions can be found in the redbook *WebSphere Studio and VisualAge for Java Servlet and JSP Programming, SG24-5755-00*

1.5 Enterprise Java Server (EJS) Runtime

The Enterprise Java Server (EJS) Runtime provides support for the Enterprise Java beans (EJB) programming model in which enterprise beans are managed by containers. Containers, in turn are executed within servers which are operating system processes that contain their own Java Virtual

Machine (JVM). Each JVM is managed by the System Management (SM) infrastructure of the application server.

The SM infrastructure allows the execution environment to be defined, enabled and monitored. The execution environment is made up of beans, containers and servers.

For more detailed descriptions and examples on each of the EJS functions please refer to 2.1, "IBM WebSphere Application Server components" on page 31.

1.5.1 Enhancements to the IBM extensions required for the EJS

The EJS programing model utilizes the RMI/IIOP model to provide distribution for the EJBs in standard and advanced releases of the EJS. The enterprise release of EJS requires the use of Interface Definition Language (IDL) to allow interoperability with the Component Broker (CB).

Some of the existing IBM extensions in the IBM Java ORB have been re-implemented as an effort by the Component Broker team to enhance the JBroker Java ORB to support the EJS. Some of these features are briefly discussed below.

1.5.1.1 Persistent Name Service (PNS)
PNS is not an ORB feature but it is required to provide reference to the persistent objects. PNS implements CORBA CosNaming specification and this mechanism will be standardized for pluggable persistence.

1.5.1.2 Object Resolver
The Object Resolver provides a pluggable interface to allow an external class to act as a specialized object adapter.

1.5.1.3 Request Interceptor (RI)
The RI allows access to the request header after marshalling out and before demarshalling in. This was done because the Request and Message Interceptor design from the original Sun ORB was not compatible with how C++ ORB handled interceptors.

1.5.1.4 Property setting and getting flags
Since the OMG defines only a couple of basic properties (`org.omg.CORBA.ORBClass, org.omg.CORBA.ORBSingletonClass`), they are not enough for IBM-specific properties. Support for these properties has been added and it will affect the method `ORB.set_parameters` and other affiliated methods.

1.5.1.5 JNDI support

EJS implements the Java JNDI APIs to support the CORBA CosNaming.

1.5.1.6 Java Transaction Service (JTS)

JTS supports the ORB. This is done by implementing the Request Interceptors, which allows the JTS to be enabled by the ORB.

1.5.1.7 Browser callbacks

Normally, a client makes method calls to an object residing on the server. This is done when the server creates a listener socket and the client opens a port to that socket. By using the same listener socket, created by the server, and the port, opened by the client, functionality has been added for the server to make method calls to the object residing in the client. The roles have been essentially reversed. This is possible only in the case of signed applets. A similar capability has been added to the C++ ORB.

1.5.2 New features in Java ORB required for the EJS

Some of the new features in the IBM Java ORB, that are required by the EJS runtime have been briefly discussed below.

1.5.2.1 Pluggable feature framework

In WebSphere V3.0, special emphasis has been paid to providing a very pluggable component-based framework. This kind of framework will take care of problems arising from receiving frequent updates of the Java ORB from Sun Microsystems and it will meet the needs of different internal customers with varying requirements. This feature is meant for private use only by the ORB team

1.5.2.2 Pluggable transport layer

In order to support any environment, it is necessary to have a pluggable transport layer. EJS has a pluggable transport layer, which means that the socket creation will have to take place in this layer. Other ramifications are that the CDRInputStream and CDROutputStream will also have to be replaceable.

1.5.2.3 SSL support

Support for the SSL interface on the server side has been provided. Using the MIME encoded IIOP allows browsers and Web servers to use SSL. By selecting to enable SSL, the user can force an SSL connection between the client and Web server for greater protection of the user ID and password data.

1.5.2.4 Mime-encoded IIOP tunneling with HTTP

IIOP tunneling means sending IIOP messages embedded in an HTTP request/response. When the client or a firewall does not allow a post method, the mime encoded IIOP tunneling technique can be used. This technique allows the mime encoded IIOP to tunnel through an HTTP firewall or Web server as an HTTP message.
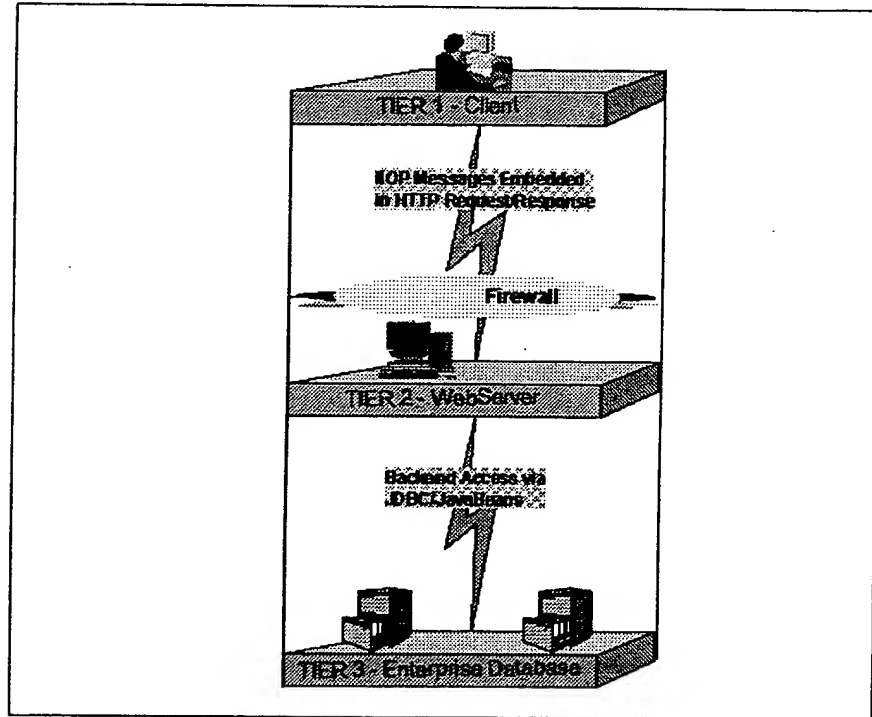
Figure 10. Mime encoded IIOP tunneling through a 3-tier framework

1.5.2.5 IIOP tunneling

There are 2 types of IIOP tunneling. In the first approach, multiple IIOP messages are embedded in an HTTP request and are communicated to the ORB using sockets. This approach has some security implications on Web servers and will not be used. The second approach is to attach the IIOP request to a single HTTP request as a parameter with binary data. This request reaches the ORB using the servlet that is started by the HTTP request. The response is then sent back as binary IIOP data. This approach is used since it is more secure.

1.5.2.6 IIOP firewall support

SOCKS V5 is used to provide the IIOP firewall support by providing authenticated connections.

1.5.2.7 IIOP redirector

IIOP redirector works in a similar fashion to the proxy firewall. It copies any replies/requests on a socket to a socket on which the ORB is connected.

1.5.2.8 Configurable requests time-outs

Only client-side timeouts have been implemented. This is done by throwing the NO_RESPONSE system exception on the client side with the completion status of *maybe*. The timeouts are set by using two ORB properties:

1. com.ibm.CORBA.RequestTimeout

2. com.ibm.CORBA.LocateRequestTimeout

The default value for both of these timeouts is 0. A timeout value of 0 means an infinite timeout interval.

1.5.2.9 eNetwork Dispatcher

The eNetwork Dispatcher is a WebSphere HTTP sprayer used from Work Load Management (WLM).

1.5.2.10 LDAP naming service

EJS provides the LDAP naming service support indirectly through a JDBC layer implemented for CosNaming. As part of LDAP support, the EJS also supports the URL context.

1.5.2.11 Work Load Management support (WLM)

WLM distributes the workload or requests across a server group. This functionality is supported in WebSphere V3. Smart proxies are used within the client along with a WLM runtime.

1.6 Preparing for Installation - What to change and why

Before running the setup of WebSphere for version 3.0 it was necessary to make some of the following changes, to ensure a good install. These hints were documented in the WebSphere Readme file that shipped with the product

1.6.1 Set the JAVA_HOME environment variable

Setting this variable allowed the OLT install to find the correct files as discussed in 4.3.0.2, "Installing and configuring the OLT/OLD environment" on page 268. The 3.02 WebSphere install on NT completes successfully even if the java_home variable is not set.

For the Windows NT platform:

1. Open the Control Panel.

2. Double click the **System** icon.

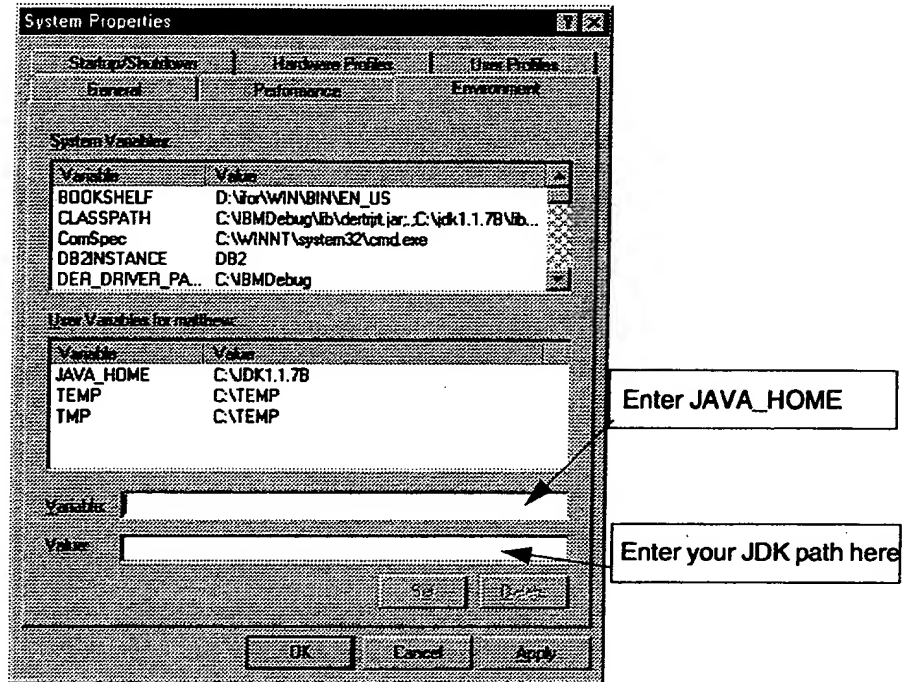3. Select the **Environment** tab.

Figure 11. The Windows NT System Properties Environment settings

4. Type in JAVA_HOME in the variable field.

5. Type in your JDK path, for example c:\JDK1.1.7B in the value field.

6. Click **Set.**

7. Click **OK.**

For the AIX Platform:

1. Log in as root or a super user.

2. cd /etc

3. Edit the environment file with an editor of choice, for example, vi.

4. Add the line JAVA_HOME=/usr/jdk_base.

5. Save the file and exit. Next time you log in the environment will be updated. Below is an example of our settings within the /etc/environment file:

```
PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin
TZ=EST5EDT
LANG=en_US
LOCPATH=/usr/lib/nls/loc
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
LC_FASTMSG=true
# ODM routines use ODMDIR to determine which objects to operate on
# the default is /etc/objrepos - this is where the device objects
# reside, which are required for hardware configuration
ODMDIR=/etc/objrepos
# IMNSearch DBCS environment variables
IMQCONFIGSRV=/etc/IMNSearch
IMQCONFIGCL=/etc/IMNSearch/dbcshelp
# Added by Steen
JAVA_HOME=/usr/jdk_base
LD_LIBRARY_PATH=/usr/jdk_base/lib/aix/native_threads:/usr/WebSphere/AppServer/pl
ugins/aix:/home/db2inst1/sqllib/lib:/usr/lib
```

Figure 12. /etc/environment settings

1.6.2 Increase the DB2 application heap size for the WAS database

In order for the Admin Server to work correctly the application heap size must be changed from its default setting of 128 to a new value of 256. In the 3.02 install on NT the installation will update this setting automatically for you. This can be done manually either through the DB2 UDB Control Center or through the db2 command line interface, below are examples of both.

From the DB2 UDB Control Center:

1. Open the Control Center and expand the system that the WAS Database resides on, so you can see the WAS DB on your screen, as shown below in Figure 13 on page 27:
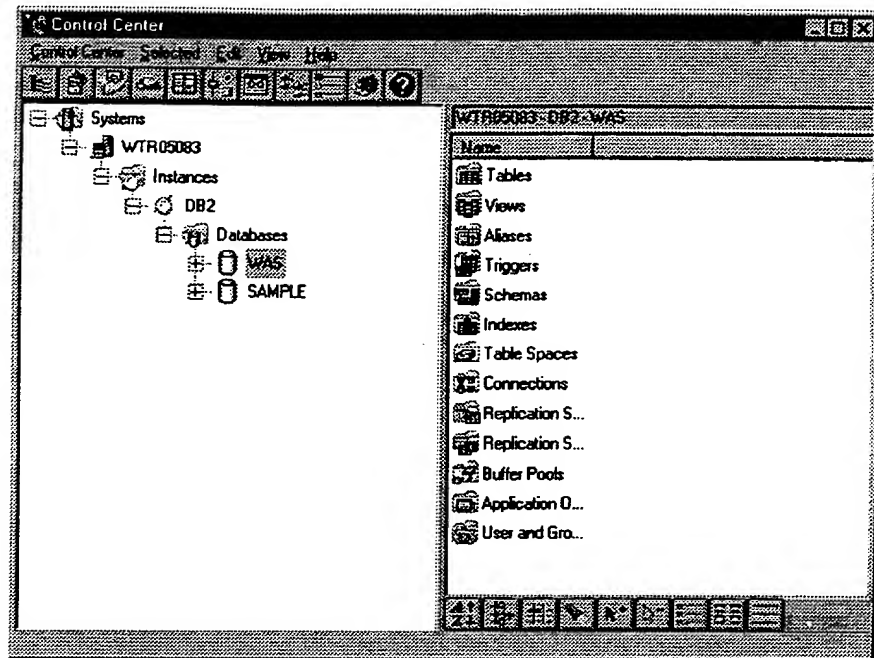
Figure 13. DB2 UDB Control Center - the WAS database

2. Right-click the **WAS** database icon and select **Configure**.
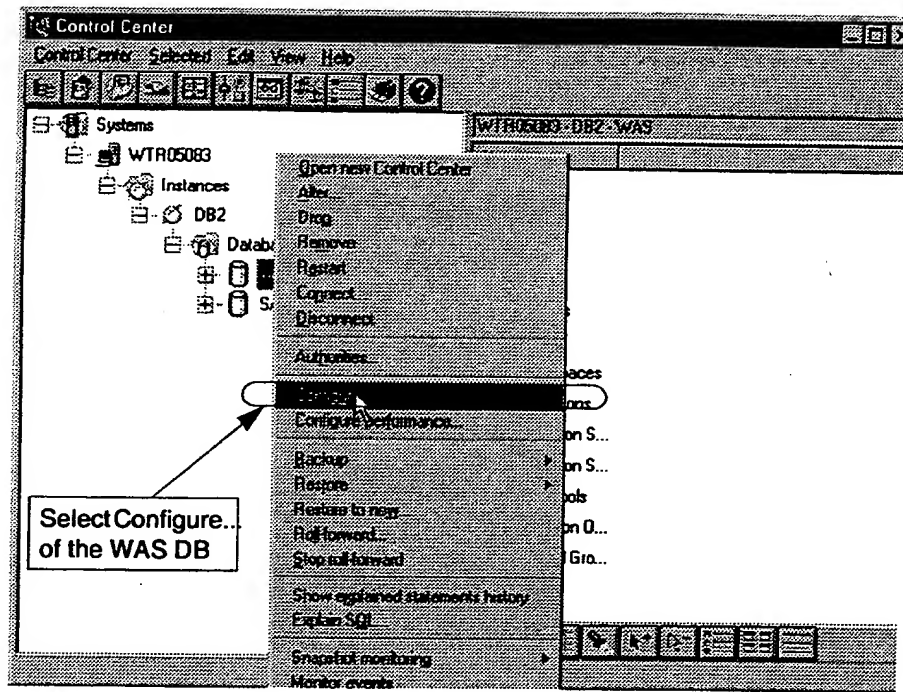
Figure 14. Configure the WAS database

3. Select the **Performance** tab.

4. Scroll down the list of parameters and select **Application heap size**.

5. Change the value from 128 to 256.
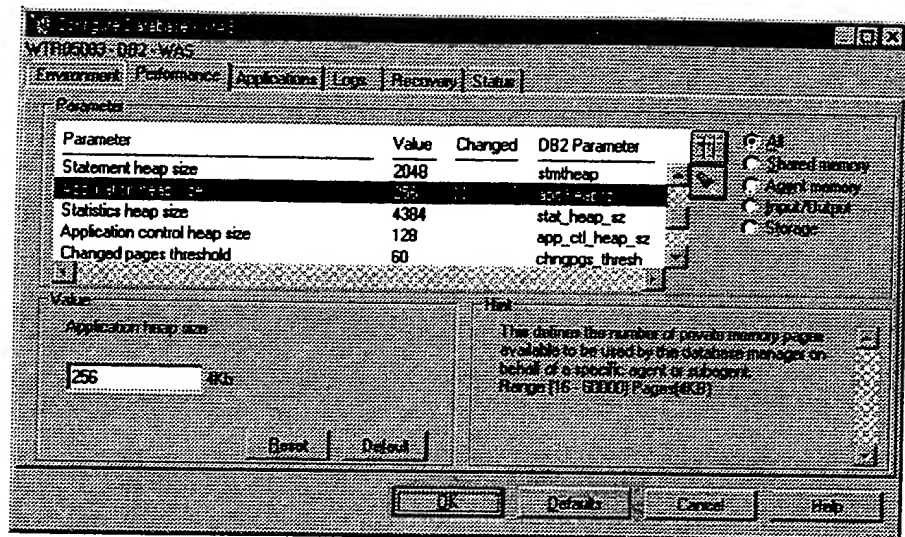
Figure 15. How to change the application heap size

6. Click **OK**.

7. Close the Control Center.

From the command line:

1. Start up a DB2 Command Line window (NT) or ensure your ~/sqllib/db2profile has been activated (AIX).

2. Connect to the WAS database with a valid user ID and password:

   ```
   connect to was
   ```

. 3. To update, type the following command:

   ```
   update db cfg for was using APPLHEAPSZ 256
   ```

4. Disconnect all applications:

   ```
   force applications all

   terminate
   ```

5. Then stop and start db2.

Chapter 2. WebSphere Application Server overview

This chapter will give a high-level overview of the Enterprise Java Server functions as deployed in IBM WebSphere V 3.0.

2.1 IBM WebSphere Application Server components

WebSphere Enterprise edition ships with two different Enterprise Java Servers as discussed in 1.3.2, "EJB architecture in brief" on page 12. The following is an overview of the main components in the EJS provided by WebSphere Advanced Edition (AE) and a brief look at the main components of the EJS provided by Component Broker (CB). We also discuss the features of WebSphere Standard Edition because they are included in Advanced Edition and they provide the necessary foundation for Advanced Edition functions.

2.1.1 Architecture overview

This section gives an overview of the components in the general EJS architecture.

Enterprise Java Services (EJS) refers to the infrastructure designed to run servlets and Enterprise Java beans.

The EJS is based on Sun's Enterprise JavaBeans Technology specifications that specify an enterprise Java platform defined through a set of standard Java APIs that provide access to existing infrastructure services.

2.1.1.1 EJS architecture

An EJB server provides the following components:

- The EJB server runtime

  The server runtime can be seen as a generic server (or model/template server) from which the Web application server instances can be modeled. EJBs live in containers that again live in the server runtime (Web application server) see Figure 18 on page 37.

  Servlets also live in a special container (servlet engine) that again live in the server runtime (Web application server) see Figure 17 on page 36.

- The EJB containers

  EJB containers are provided following the requirements as described in the Enterprise JavaBeans specifications Version 1.0 (for further information see http://java.sun.com/products/ejb/). Furthermore, IBM

provides additional features for example, entity bean support with bean managed or container managed persistency and a simple deployment tool.

See 2.1.2, "Enterprise Java beans and containers" on page 39 for more details on containers and EJBs.

• The Enterprise Java beans

This combination of components provides a number of services. These are:

• A deployment tool

When an EJB has been developed it has to be transformed into a form that enables it to be managed and accessed. The transformation is referred to as EJB deployment.

To deploy an EJB you will normally use a built-in tool of an IDE (Integrated Development Environment) or a stand alone tool.

See 2.1.4, "Deployment tools" on page 40 for more details.

• Naming services

The IBM WebSphere Application server architecture provides naming and directory services to provide an interface to find EJBs based on the name or an attached attribute.

In IBM WebSphere the Java Naming and Directory Interface (JNDI) is used to provide a common interface to the actual naming and directory service that is being used.

JNDI provides an Application Program Interface (API) to be accessed through a Java application and a Service Provider Interface (SPI) to specify the interface to existing and widely used name and/or directory services. The purpose of providing an open SPI specification is to make the JNDI independent of the specific naming or directory service implementation used.

For further information on the JNDI specifications see:
http://java.sun.com/products/jndi/

For more details on naming and directory services see Chapter 7, "WebSphere access control and security" on page 363.

• Security services

The security services provide support for Web resources (for example, HTML, JSP and CGI files), servlets and EJBs.

The security authorization information, authentication and delegation policies will typically be defined using the IBM WebSphere Administrative

Console. In WebSphere Advanced the security service is an EJB server that contains security beans.

See Chapter 7, "WebSphere access control and security" on page 363 for more details on security.

- Work Load Management

A Work load management (WLM) mechanism is provided to make scaling of enterprise applications running on IBM WebSphere Application server possible.

Workload Management is a service that improves the scalability of an EJB server runtime environment by grouping multiple servers together into a server group. EJB clients can access this server group as if it was a single server. The actual server that responds to the client request will be transparently determined by the Workload Management service. WebSphere implements a number of different policies for how the Workload Management service will choose the server.

We do not cover Workload Management in this book.

- A persistence service

This service provides support for the proper interaction between a bean and its data source to ensure that any persistent data is maintained.

In AE this is accomplished by using the JDBC API to interface with relational databases and Java transaction support.

In CB the persistent service is accomplished using the X/Open XA interface to relational databases and the OMG Object Transaction service.

- A transaction service

This service implements the transactional attributes specified in an EJB's deployment descriptor.

- System management infrastructure

The system management infrastructure enables management of Web server, Web application server and Web application resources. A client interface is provided for the administrator to manage these resources.

An overview of the AE system management infrastructure is given in 2.1.5, "System Management Infrastructure" on page 40.

The IBM WebSphere Administrative Console is provided with IBM WebSphere 3.0 Advanced.

See 2.1.5.1, "Systems management console" on page 41.

The client interface for the EJB server (CB) environment is the systems management end user interface. We do not discuss this interface in this redbook. Please refer to the redbook *WebSphere Application Server Enterprise Edition Component Broker 3.0 First Steps*, SG242033 for more details.

2.1.1.2 WebSphere Application Server Standard Edition

The WebSphere Application Server Standard Edition provides a basic Web application server environment for Web applications that typically consist of HTML files, JSP files, applets, servlets, image files and databases.

The primary purpose of the IBM WebSphere Application Servers is to provide an environment into which scalable, portable, well performing and reliable Web applications can be developed and executed based on Java-based programming, standard techniques, Internet standards, standard middleware and database management systems.

IBM WebSphere Application Server Standard Edition provides an environment where you can extend and enhance your Web applications by migrating your HTML to JSP. Since JSPs essentially are HTML files with additional JSP-specific tags you can use your HTML skills and standard development tools.

IBM WebSphere Application Server compiles the JSPs (at runtime) and transfers them into servlets. However, this process is managed by IBM WebSphere Application Server and is transparent to the developer.

Since JSPs can include Java code and you write your servlets in Java you can use Java in your development - if you want to or the requirements force you to. However, you are not necessarily required to do so.

"Traditional" Web development components like CGI programs, Web Server APIs and client side scripting languages may also be utilized since an IBM Web Application Server setup includes a "traditional" Web server.

IBM WebSphere Application Server Version 3 integrates with many different Web Servers and includes plug-ins for IBM HTTP Server, Apache, Lotus Domino Go V4.6.2.5, Lotus Domino, Netscape Enterprise Server, and Microsoft IIS.

To integrate with a Web server you must select one or more of the plug-in extensions as shown in Figure 16 on page 35.
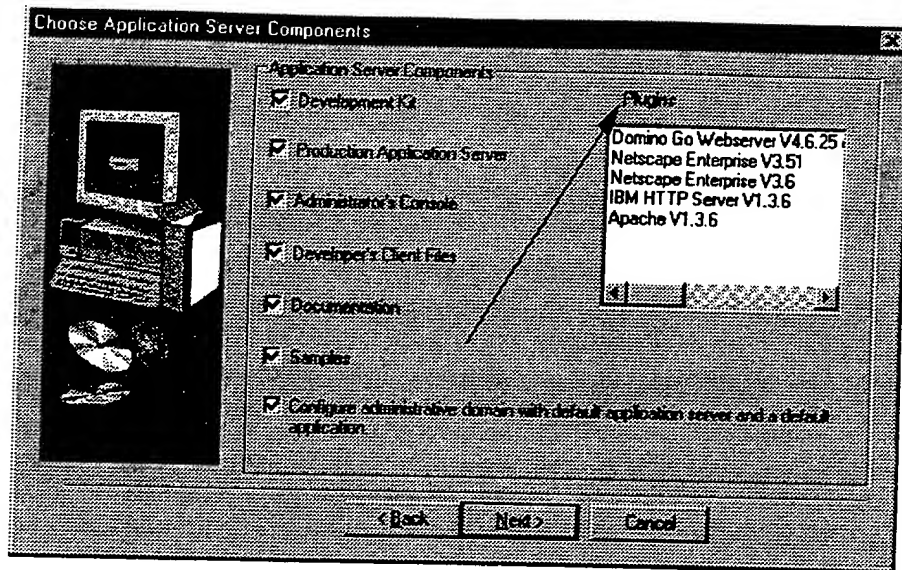
Figure 16. Web servers that integrate with the application server

The application server provides the runtime environment for execution of servlets. See Figure 17 on page 36 for a view of this runtime architecture.

The Web application server also provides a connection manager function that manages database connections. The connection manager provides an easy to use mechanism for reducing the resources required by Web applications when accessing databases.

Furthermore, the Web application server provides transaction support through an implementation of Java Transaction Server (JTS) in relation with JDBC/XA databases.

Finally, the IBM WebSphere Application Server Standard Edition architecture includes a system management infrastructure with two primary components the Administrative server and the Administrative client (console) (seeFigure 17 on page 36).

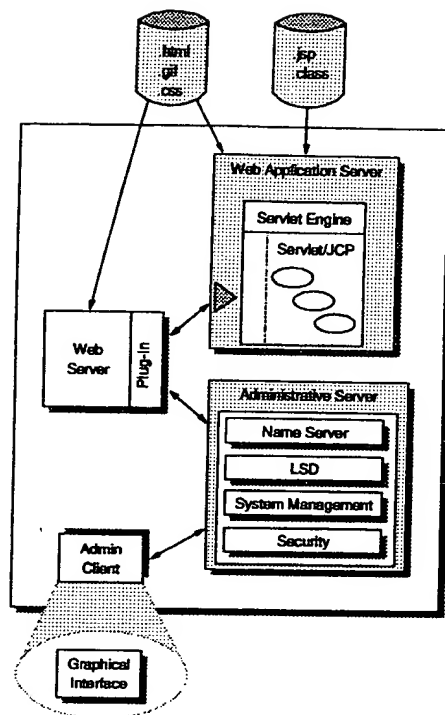For further information on the system management structure see 2.1.5, "System Management Infrastructure" on page 40.

Figure 17. WebSphere Application Server Standard Edition 3.0 runtime architecture

2.1.1.3 WebSphere Application Server Advanced Edition

The WebSphere Application Server Advanced Edition (AE) has the functions found in the WebSphere Application Server Standard Edition plus support for EJBs and distributed (clustered) systems. This application server is one of two EJB servers provided in IBM WebSphere Enterprise Edition.

Since IBM WebSphere Application Server Advanced Edition is an extension of the Standard Edition you can easily move an application developed for a server running Standard Edition to servers running Advanced Edition.

The Web application server in IBM WebSphere Application Server Advanced Edition (see Figure 18 on page 37) includes support for EJB containers (for the EJBs) besides the servlet engine (for the servlets and JSPs) also found in Standard Edition.
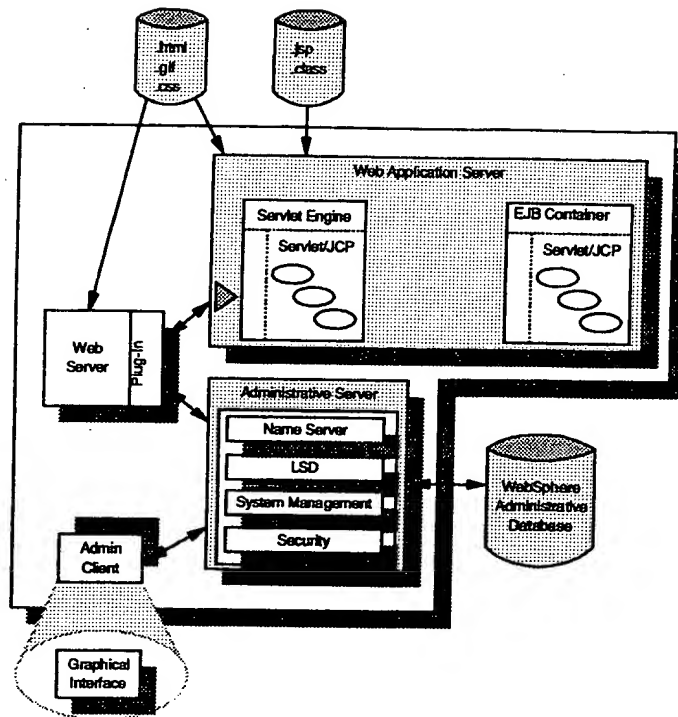
Figure 18. WebSphere Application Server Advanced Edition 3.0 runtime architecture

There is a single instance of the administrative server on a single node (physical machine).

In a multi-node setup (cell) there will be a single instance of the administrative server on each node. The administrative servers contains identical configuration information and access the same configuration repository in the same database (see Figure 19 on page 38).

The rationale is that you should be able to access any one of the administrative servers in a cell and see changes reflected on any other administrative server in the cell (single administrative image).

The WebSphere administrative database can be located either on a separate database server (physical machine) or on one of the machines that host the administrative servers. However, each machine must have database server, client or connection software installed and configured to enable access to the common database.

If you plan to run the same Web applications and enterprise applications on more than one physical machine you will also have to ensure that all systems have access to identical (or the same) files and in identical locations in the directory structures. This can be accomplished either by creating identical files and directory structures on each machine or by using a shared file system. If the first method is used we would recommend that you establish an automatic or semi-automatic procedure to ensure that the data are identical.
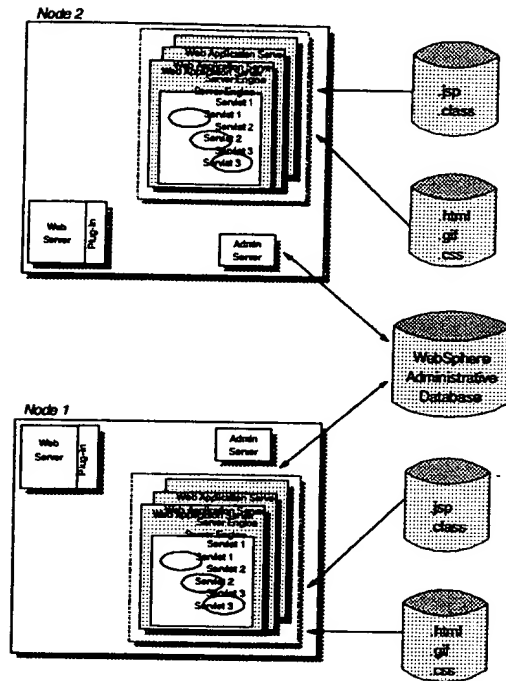
Figure 19. WebSphere Application Server Advanced Edition multi-node setup

A concept of server groups has been implemented in the IBM WebSphere Application Server for management and control purposes.

The following applies to the server group concept:

- Server groups consist of one or more Web application servers.

- A single Web application server can only belong to one server group.

- A server group can be seen as a logical server.

- Web application servers within a group are clones of each other. Clones in this context means logically identical application servers with respect to configuration of resources for example, containers and EJBs.

- The Web application servers within a server group may be distributed to different nodes (physical machines).

The server group is very useful in relation to workload management. A request can be forwarded to a server group and the request is handled by a server in the group while the specific server identity is hidden from the requester.

2.1.2 Enterprise Java beans and containers

The IBM WebSphere Application Server Version 3.0 server architecture provides a generic server (the Web application server, see Figure 18 on page 37) in which EJB containers live.

One of the primary responsibilities of an EJB container is to provide a number of fundamental services for example, transaction, state, security and persistence to the EJB. The advantage being that it reduces the work required by the EJB developers and supports development of portable EJBs.

Another responsibility of an EJB container is to create the interfaces (EJB Home and EJB Object) required for an EJB client to access a deployed EJB so the EJB client interacts indirectly with the EJB through the EJB container.
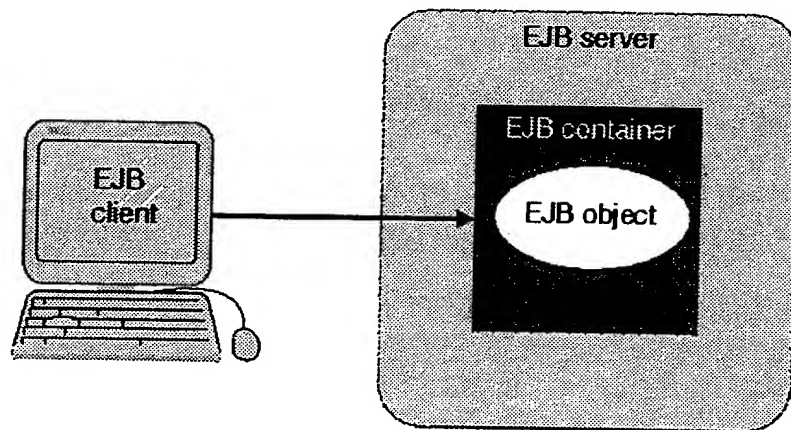

Figure 20. Client - EJB container relationship

2.1.3 Servlets and the application server

A special container (a servlet engine) is provided in IBM WebSphere Application Server to enable servlets to be run within the application server.

The servlet engine has been designed to be integrated as seamlessly as possible in the EJS architecture as well as in the security and system management infrastructure. The design approach allows for consistent system management for servlets and EJBs as appropriate while maintaining the typical servlet environment and characteristics intact.

Servlets are created and managed as members of a Web application in IBM WebSphere Application Server Version 3.0. IBM WebSphere Application Server provides work load management support for servlets (as it does for EJBs) to allow requests for servlets within a server group to be distributed to application servers belonging to the group.

2.1.4 Deployment tools

When you have developed your Enterprise Java beans for example, using VisualAge for Java or manually, they have to be deployed.

When you deploy an EJB you create or modify a Jar file which includes a description of the Jar file contents (the manifest), deployment descriptors, bean class files and potentially environment properties.

To create a deployed EJB Jar file you can for example, use VisualAge for Java, the Jetace tool or you can use the tools available in the Java Development Kit (JDK). The Jetace tool is provided as part of the IBM WebSphere Application Server.

After the EJB Jar file has been deployed for your IBM WebSphere Application Server it must be installed in your environment in accordance with the WebSphere administrative infrastructure. IBM WebSphere Application Server Advanced Edition provides the WebSphere Advanced Administrative Console to be used for EJB creation (installation).

Depending on the requirements for your EJB you may even choose to deploy an undeployed Jar file during EJB creation using the WebSphere Advanced Administrative Console.

You will also find a command line tool wlmjar with WebSphere Application Server Advanced Edition that can be used to create a workload-managed prepared version of your Jar file.

2.1.5 System Management Infrastructure

This section describes the IBM WebSphere Application Server administration model.

2.1.5.1 Systems management console

The WebSphere Administrative Console is the system management console for IBM WebSphere Application Server Version 3.0 Advanced Edition.

An administrative domain (sometimes referred to as a cell or sphere,) is a collection of managed nodes, that is, host machines. Each managed node has an administration server executing on that node that is responsible for configuring, monitoring, and managing the WebSphere servers that run on that node. A client graphical user interface is provided to enable the administrative domain to be defined.
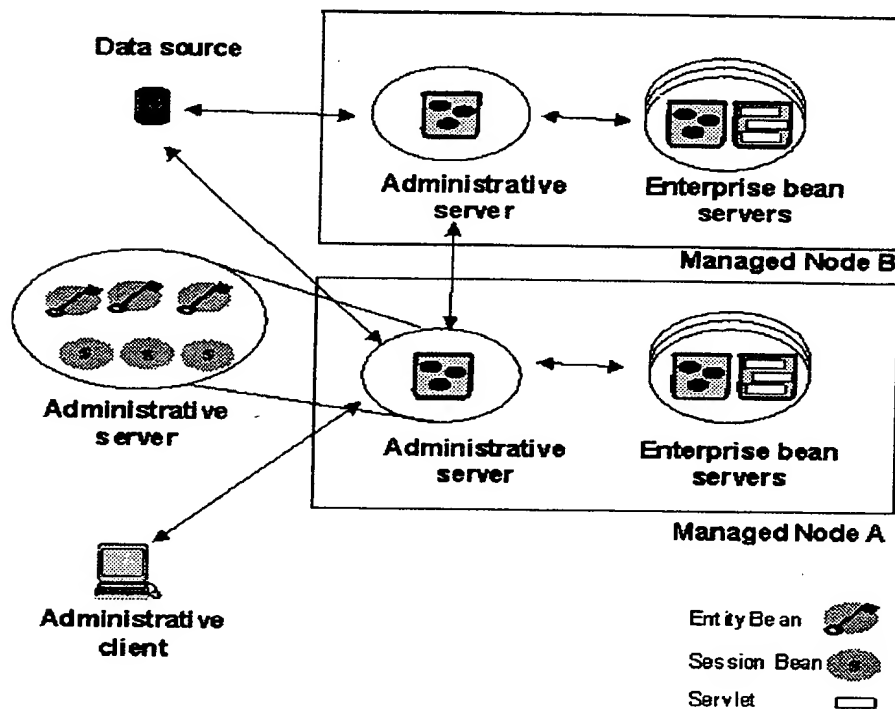

Figure 21. WebSphere Administration Server administration model

This Graphical User Interface (GUI) makes it easy to interact with the beans that were loaded by the adminserver. The front-end has a tab look and feel and has three main tabs as shown in Figure 22 on page 42, Figure 23 on page 43, and Figure 24 on page 45. The WebSphere Administrative Console

has three tab panes namely Types, Topology, and Tasks. These three panes are discussed below.

2.1.6 The Types view

The Types view is a hierarchical view of all resources on all nodes in the administrative domain. Each folder icon represents a different resource type. By selecting any object and right clicking, a context-sensitive menu appears. This menu has the basic tasks such as Create, Move, Default Properties.

The Types pane is shown in Figure 22 on page 42.
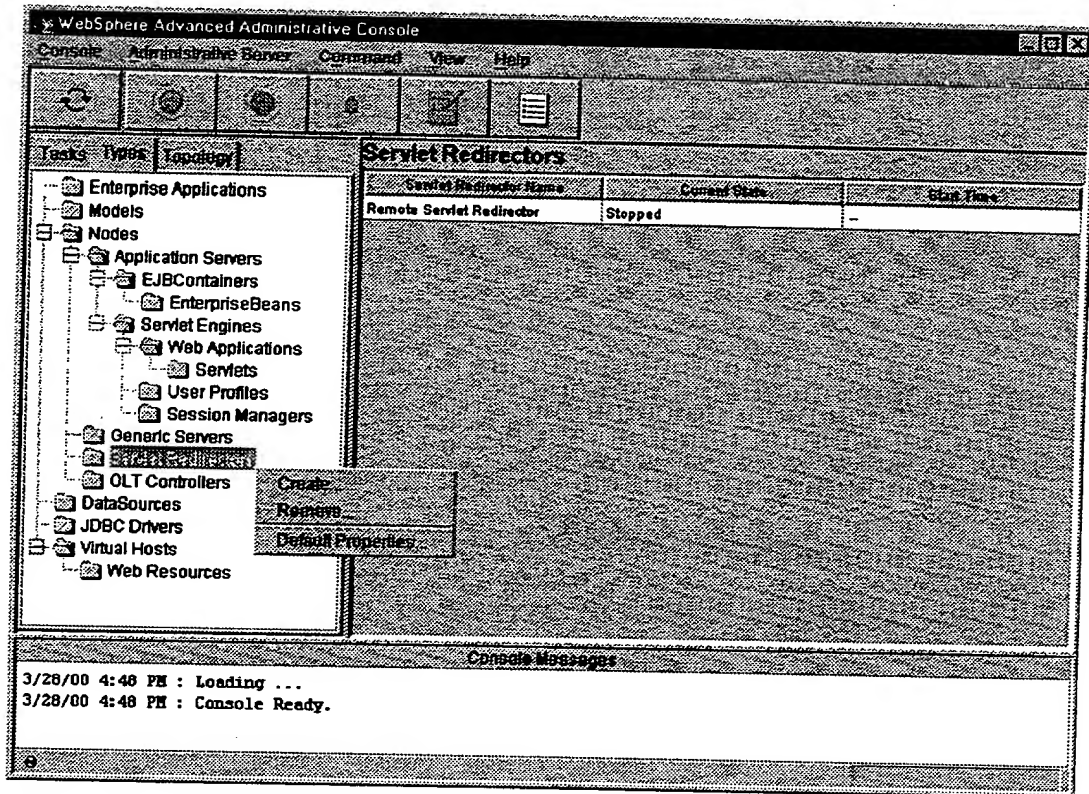
Figure 22. WebSphere Advanced Administrative Console - Types tab

2.1.7 The Topology view

The Topology pane consists of the WebSphere Administrative Domain and all the managed nodes as shown in Figure 23 on page 43. For each managed

node there is an associated hierarchy of all the resources within that node. The resource attributes can be changed and methods can be invoked on these resources in this view also, just as in the Types view.
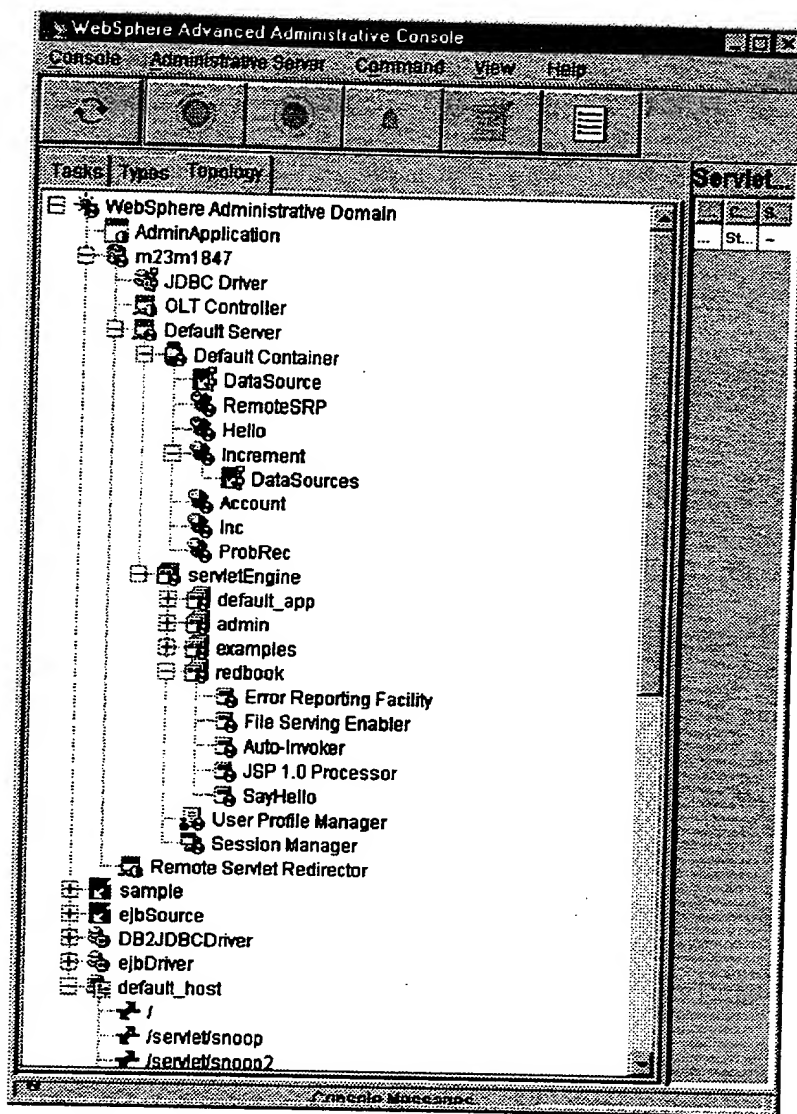
Figure 23. WebSphere Advanced Administrative Console - Topology tab

2.1.8 The Tasks view

The Tasks view shows different tasks that can be performed and is shown in Figure 24 on page 45. It consists of three task groups, which are briefly discussed below.

2.1.8.1 Configuration tasks

This group of tasks allows you to configure application servers, virtual hosts, servlet engines, web Applications, and enterprise applications.

2.1.8.2 Performance tasks

This task allows the user to launch the Resource Analyzer tool to monitor performance and load statistics. The Resource Analzer is dicussed in more detail in the redbook *WebSphere V3 Performance Tuning Guide*, SG24-5657-00.

2.1.8.3 Security tasks

Using this task, you can apply security to enterprise applications and to the HTTP and EJB methods of resources such as servlets and enterprise beans.
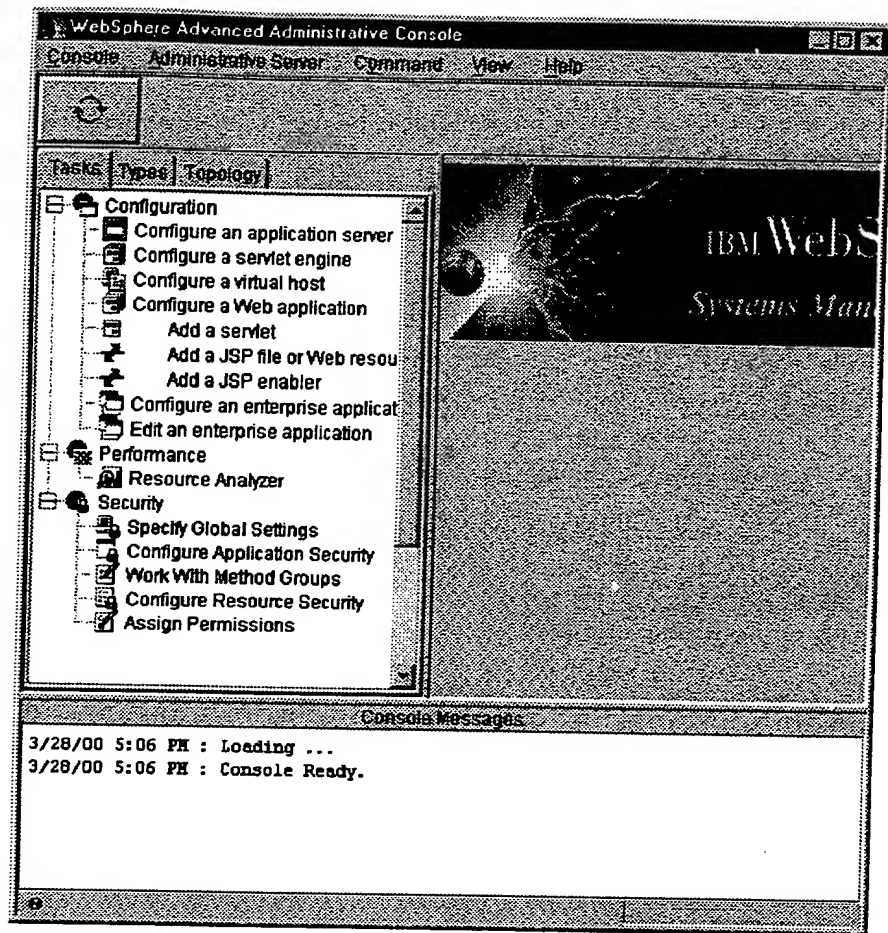
Figure 24. WebSphere Advanced Administrative Console - Tasks tab

8.1.6 Starting the LDAP directory server from the command line

To start LDAP from the command line you should use the following command:

`/usr/ldap/bin/slapd`

8.1.7 Administration interface - Web

From the Directory Server Web Admin you can perform several administrative task such as:

- Configure a database
- Configure a replica
- Start up and shut down the server
- Define an ACL
- Add and delete suffixes
- Add entries to the directory

To use the Directory Web Admin you have to start up a Web browser and type in the Location field `http://hostanme/ldap`
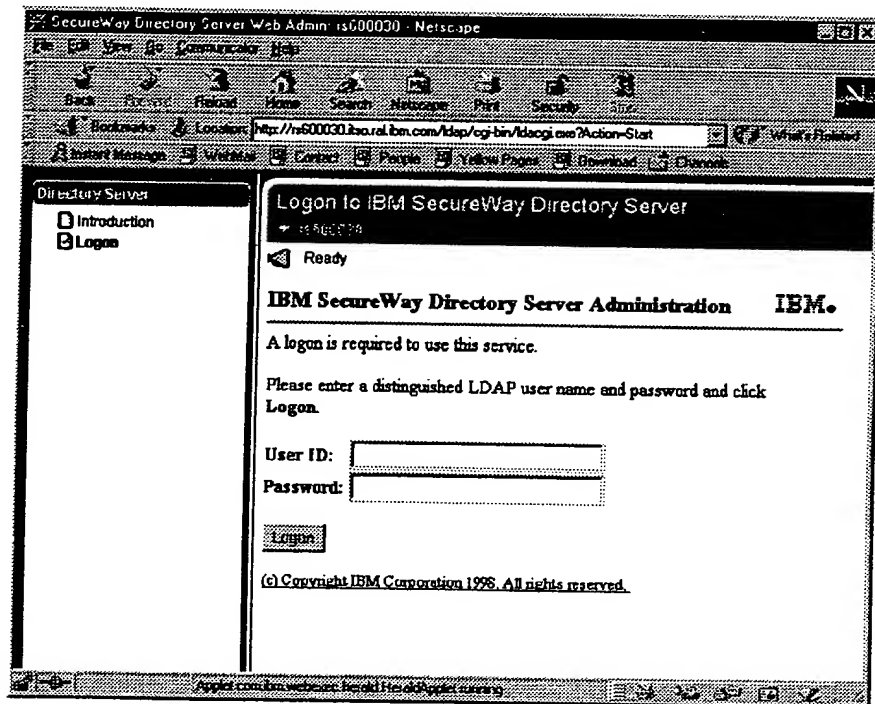
Figure 405. ldap Web admin
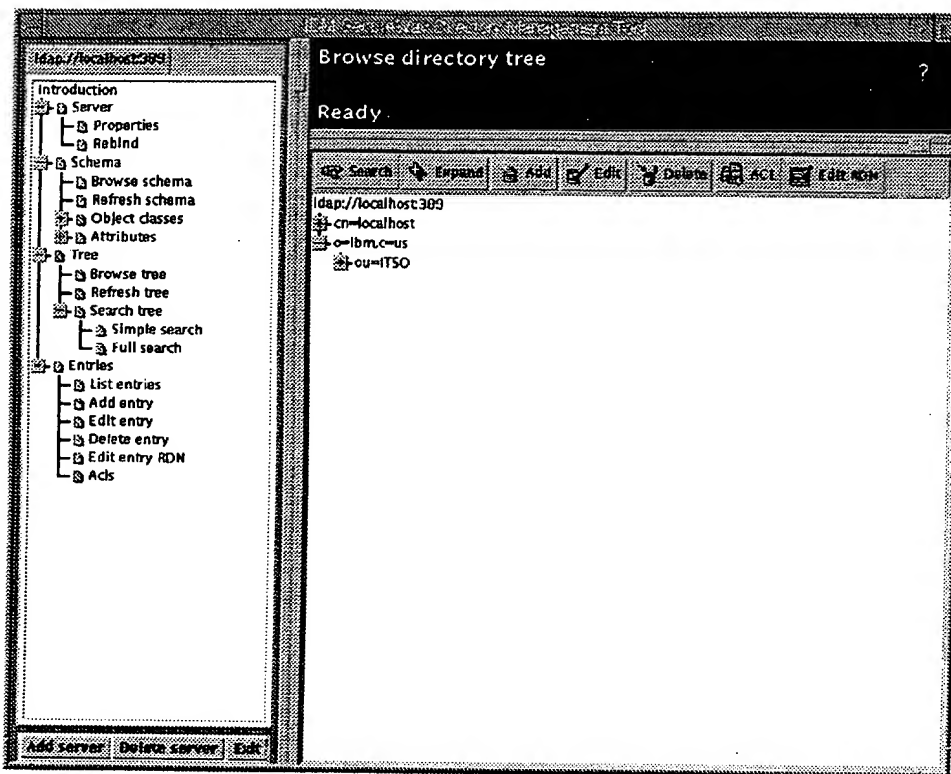
Figure 406. DMT

For more information about how to use this tool, see:

http://www-4.ibm.com/software/network/directory/

8.1.9 Configuration files - attributes and objects classes

The file /etc/slapd.conf contains configuration properties such as the SSL port, DB2 user, admin DN, and the admin password.

```
adminPW
>14II9OIR3WkRGgTEK+GHehiE31zkF1XEuvM3UXxkZ1Gm9KhP8FPcLFm4nBKB9fC2bRyCUARkQNikIPJalNHS9al9KdYn3it
M4Vj2xFmh/jFU2V0b7Bf2kTVvA+FSn1oyo8WVYolgU59+Z1jH458RneygaJdkz8iX2R<
adminDN    "cn=root"
# IBM SecureWay Directory Server Configuration File V3.1.1 for AIX
# CAUTION: EDIT THIS FILE WITH CARE
# We recommend making all changes through the server administration interface.
sysLogLevelm
```

```
ldapmodify [options] [-f <ldif input file>]

ldapadd [options] [-f <ldif input file>]
```

Some options are:

- -h host LDAP server host name
- -p port LDAP server port number, default 389
- -D dnbind dn by default anonymous
- -w   bind password
- -R specifies that referrals are not to be automatically followed
- -M manage referral objects as normal entries
- -V LDAP protocol version (2 or 3; default is 3)
- -C charset character set name to use, as registered with

                IANA

- -b Assumes that any values that start with a / are binary values, and that the actual value is in a file whose path is specified in the place where values normally appear
- -c   continuous operation; do not stop processing on error
- -n   show what would be done but don't actually do it
- -v   verbose mode
- -d level   set debug level in LDAP library

Examples

Following are some examples using the LDAP commands:

```
ldapadd -h rs600030 -D "cn=root" -w swall7r -f add.ldif
```

This is the add.ldif's contents:

```
cn=Marisa,ou=ITSO,o=ibm,c=us
sn=cicsMan
objectclass=ePerson
objectclass=person
objectclass=inetOrgPerson
objectclass=top
objectclass=organizationalPerson
uid=Marisa
mail=Marisa@ar.ibm.com
cn=Marisa
userPassword=swall7r
```

Now let's modify the Marisa's mail attribute:

```
ldapmodify -h rs600030 -D "cn=root" -w swall7r -d mod.ldif
```

```
cn=Marisa,ou=ITSO,o=ibm,c=us
objectclass=ePerson
objectclass=person
objectclass=inetOrgPerson
objectclass=top
objectclass=organizationalPerson
mail=cicsMan@ar.ibm.com
```

To validate the change we ran the ldapsearch command:

```
ldapsearch -h rs600030 -b "ou=ITSO,o=IBM,c=US" cn=Marisa
```

```
cn=Marisa,ou=ITSO,o=ibm,c=us
sn=cicsMan
uid=Marisa
cn=Marisa
objectclass=ePerson
objectclass=person
objectclass=inetOrgPerson
objectclass=top
objectclass=organizationalPerson
mail=cicsMan@ar.ibm.com
```

8.1.10.3 The ldapsearch utility

The ldapsearch utility is a command-line utility built around the ldap_search() API. The utility opens a connection to an LDAP server, binds to the server, and performs a search using a specified search filter. If the request finds one or more entries, the requested attributes are retrieved, and the entries and values are printed to standard output. If no attributes are specified, all attributes associated with each returned entry are displayed. The syntax is:

```
ldapsearch [options] filter [attributes]
```

The significant parameter options for the ldapsearch tool are:

- -h LDAP server host name
- -p LDAP server port number
- -D bind dn
- -w bind password
- -b base dn for search; LDAP_BASEDN in environment is default
- -s search scope (base, one, or sub)

- -a how to reference aliases (never, always, search, or find)
- -l time limit (in seconds) for search
- -z size limit (in entries) for search
- -f perform sequence of searches using filters in file
- -A retrieve attribute names only (no values)
- -R do not automatically chase referrals
- -M manage referral objects as normal entries
- -V LDAP protocol version (2 or 3; default is 3)
- -C character set name to use, as registered with IANA
- -B do not suppress printing of non-ASCII values
- -L print entries in LDIF format (-B is implied)
- -F print separation between attribute names and values
- -t write values to files in /tmp
- -n show what would be done but don't actually do it
- -v run in verbose mode
- -d set debug level to level in LDAP library

The search filter, in many cases, will either be a simple attribute search (such as cn=Smith) or for all attributes (cn=*). Search filters, however, can be fairly complex, and there is a separate RFC (RFC 2254) that you should refer to if you need all the details. The following is a brief description of search filters. A search filter defines criteria that an entry must match to be returned from a search. The basic component of a search filter is an attribute value assertion of the form:

```
attribute operator value
```

For example, to search for a person named John Smith, the search filter would be cn=John Smith. In this case, cn is the attribute, = is the operator, and John Smith is the value. This search filter matches entries with the common name John Smith. Table 9 on page 424lists the operators for search filters.

Table 9. Operators

Operator	Description	Example
=	Returns entries whose attribute is equal to the value.	cn=John Smith finds the entry with the common name John Smith.
>=	Returns entries whose attribute is greater than or equal to the value.	sn>=smith finds all entries from smith to z*.
<=	Returns entries whose attribute is less than or equal to the value.	sn<=smith finds all entries from a* to smith.
=*	Returns entries that have a value set for that attribute.	sn=* finds all entries that have the sn attribute.
~=	Returns entries whose attribute value approximately matches the specified value. Typically, this is an algorithm that matches words that sound alike.	sn~= smit might find the entry "sn=smith".

The "*" character matches any substring and can be used with the = operator. For example, cn=J*Smi* would match John Smith and Jan Smitty. Search filters can be combined with Boolean operators to form more complex search filters. The syntax for combining search filters is:

("&" or "l" (filter1) (filter2) (filter3) ...)

("!" (filter))

The Boolean operators are listed in the following table:

Table 10. Operators

Boolean operator	Description
&	Returns entries matching all specified filter criteria.
l	Returns entries matching one or more of the filter criteria.
!	Returns entries for which the filter is not true. This operator can only be applied to a single filter. (!(filter)) is valid, but (!(filter1)(filter2)) is not.

Examples:

1. Retrieve all the entries with a person object defined;

   ```
   ldapsearch -h rs600030 -b "o=IBM,c=US" objectclass=person
   ```

2. Retrieve all the entries with an e-mail ending in "ibm.com";

   ```
   ldapsearch -h rs600030 -b "o=IBM,c=US" mail=*ibm.com
   ```

3. Retrieve all the entries with cn=Marina and mail=*ibm.com;

   ```
   ldapsearch -h rs600030 -b "o=IBM,c=US"
   "(&(cn=Marisa)(mail=*ibm.com))"
   ```

   Note that we used " " in the filter. The reason was that on AIX the ksh preprocesses the () directive with " " and doesn't preprocess the contents.

4. Retrieve all entries with mail=*ibm.com and cn not equal to Karina, root;

   ```
   ldapsearch -h rs600030 -b "o=IBM,c=US"
   "(&(mail=*ibm.com)(!(|(cn=Karina)(cn=root))))"
   ```

8.1.10.4 The ldapdelete utility

The ldapdelete utility is built around the ldap_delete() API. The utility opens a connection to an LDAP server, binds to the server, and deletes one or more entries. The distinguished names (DNs) of the entries to delete are read from standard input or from a file.

The syntax is:

```
ldapdelete [options] [DNs]
```

```
ldapdelete [options] [-f file]
```

where:

dn: one or more items to delete

file: name of input file containing items to delete

If neither dn or file is specified then items are read from standard input. The options are:

-h LDAP server host name

-p LDAP server port number

-D bind dn

-w bind password

-Z use a secure ldap connection (SSL)

-K file to use for keys

-P keyfile password

-N private key name to use in keyfile

-m perform SASL bind with the given mechanism

-R do not chase referrals

-M Manage referral objects as normal entries

-O maximum number of referrals to follow in a sequence

-V LDAP protocol version (2 or 3; default is 3)

-C character set name to use, as registered with IANA

-c continuous operation; do not stop processing on error

-n show what would be done but don't actually do it

-v verbose mode

-d level (set debug level in LDAP library

Examples:

1. Delete the entries with cn=Karina:

```
ldapdelete -h rs600030 -D "cn=root" -w swal17r "cn=Karina,
ou=ITSO,o=IBM,c=US"
```

2. Delete two entries from a file called ldapDelete:

```
"cn=Marisa,ou=ITSO,o=IBM,c=US"
"cn=Cristian,ou=ITSO,o=IBM,c=US"
```

```
ldapdelete -h rs600030 -D swal17r -f ldapDelete
```

8.1.11 Configuring the Netscape address book to use LDAP

The Netscape address book is an LDAP-enabled application, which means that it can get information from an LDAP directory. Other products like Microsoft Internet Explorer are LDAP enabled too.

In these examples we show you how to set up the Netscape address book to use IBM SecureWay Directory V3.1.1.

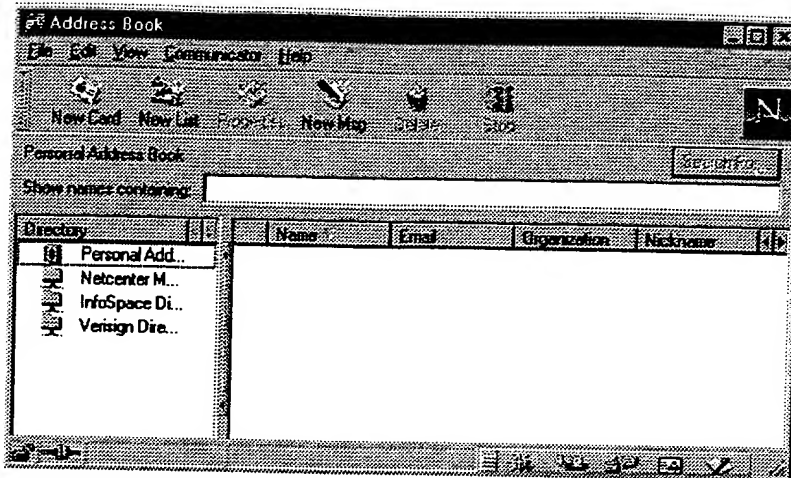1. Start the Netscape address book.

Figure 407. Netscape address book

2. Create a new directory.

Click **File -> New Directory** and you should see the dialog box shown in Figure 408.
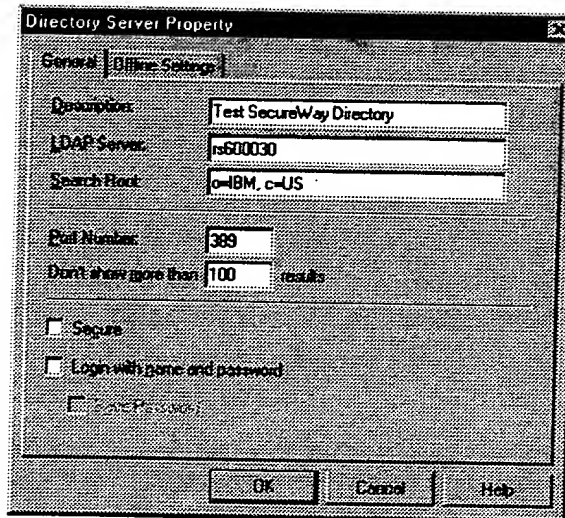

Figure 408. Directory Server Property dialog box

The fields are:

- Description: Title description is optional
- LDAP Server: Host name of the Directory Server
- Search Root: Base DN
- Port Number: By default 389
- Don't show more than: Maximum number of entries returned
- Secure: Use SSL
- Login with name and password: Used to bind with user ID and password

3. To see all entries, type * in the Show names containing field:
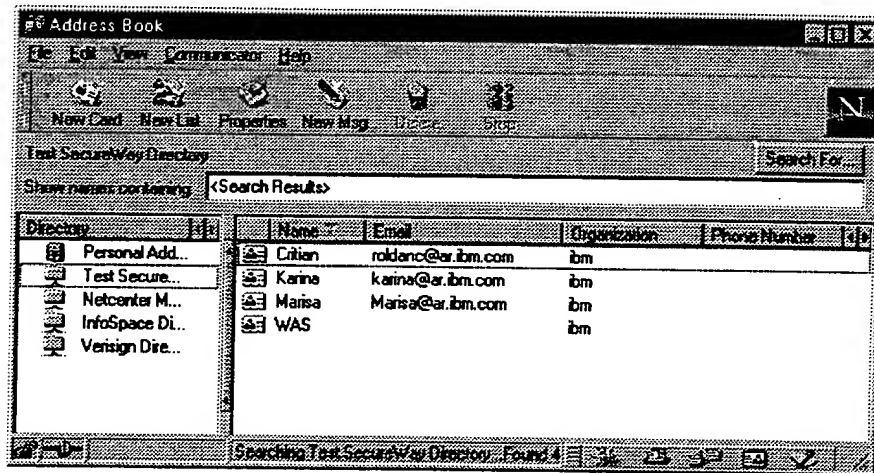
Figure 409. Search

8.1.12 WebSphere and LDAP

WebSphere supports authentication mechanisms based on validating credentials such as user ID and password, certificates, or tokens. Credentials are verified against a user registry supporting such a schema. For example, user IDs and password-based authentication can be based on the LDAP user registry where authentication is performed using an LDAP bind. A certificate validation list may be used in cases where authentication of the user is based on the client certificate presented by the user over a mutual SSL connection. WebSphere supports a three-party authentication schema, one in which the client principal and server principal are authenticated to a mutually trusted third-party. The third-party in this case is the authentication token server. An advantage of a three-party schema is that administration of the user registry can be controlled centrally.

WebSphere supports the following list of LDAP Directory Servers:

- Netscape Directory Server Version 3.x and 4.x
- SecureWay Directory Server Version 2.1 and 3.1.1
- Lotus Domino Version 4.6 and 5.0

Additional attributes are available for customizing any of the default filters to fit a Directory Server not listed above.

8.1.13 Configuring WebSphere to use LDAP

Start the WebSphere Advanced Administrative Console.

1. Select the **Tasks** tab, double-click the **Security** option and you will see a window similar to Figure 410:
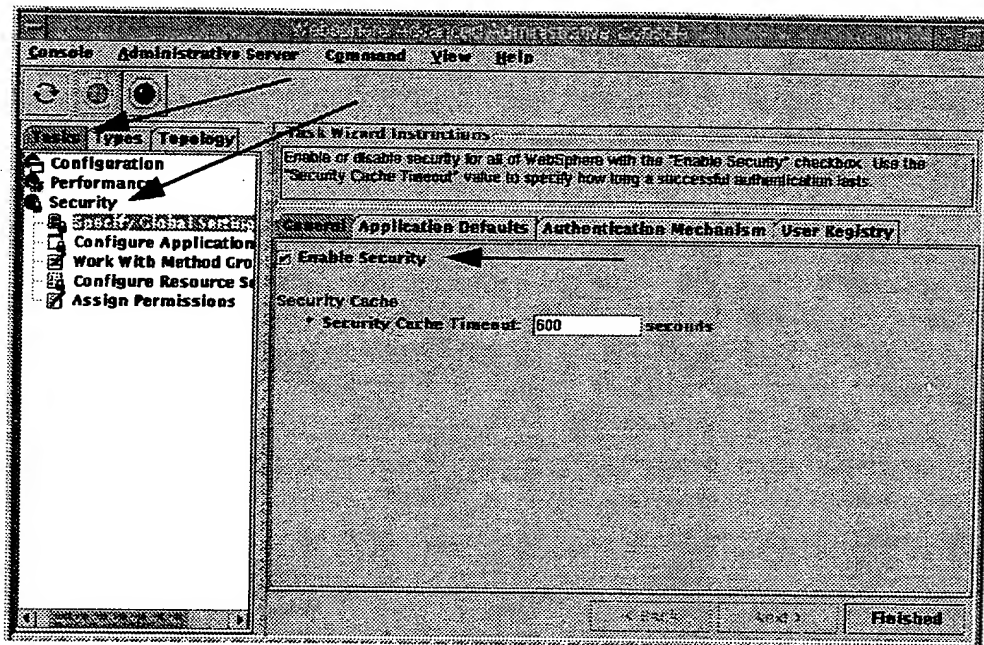

Figure 410. Security global settings

2. Select **Specify Global Settings** and click the **Enable Security** field.

    Enable Security: Specifies whether to enable or turn off server security. If you deselect this field all the security options specified on resources will be unprotected.

Security Cache TimeOut: Specifies how many seconds the server should cache security information received from the user registry (Operating System registry or Directory Server "LDAP").

3. Select the **Application Defaults** tab to specify the following properties:

Realm Name: Is the security domain where the user will be authenticated? If the principal tries to access a resource in a different realm, the principal will be prompted to log in to the new realm. It is used if Single Sign On (SSO) is used.

Challenge Type: This option specifies the mechanism used by the principal to interchange credentials. If you select **Basic**, it means that the Web browser will prompt the principal for a user ID and password.
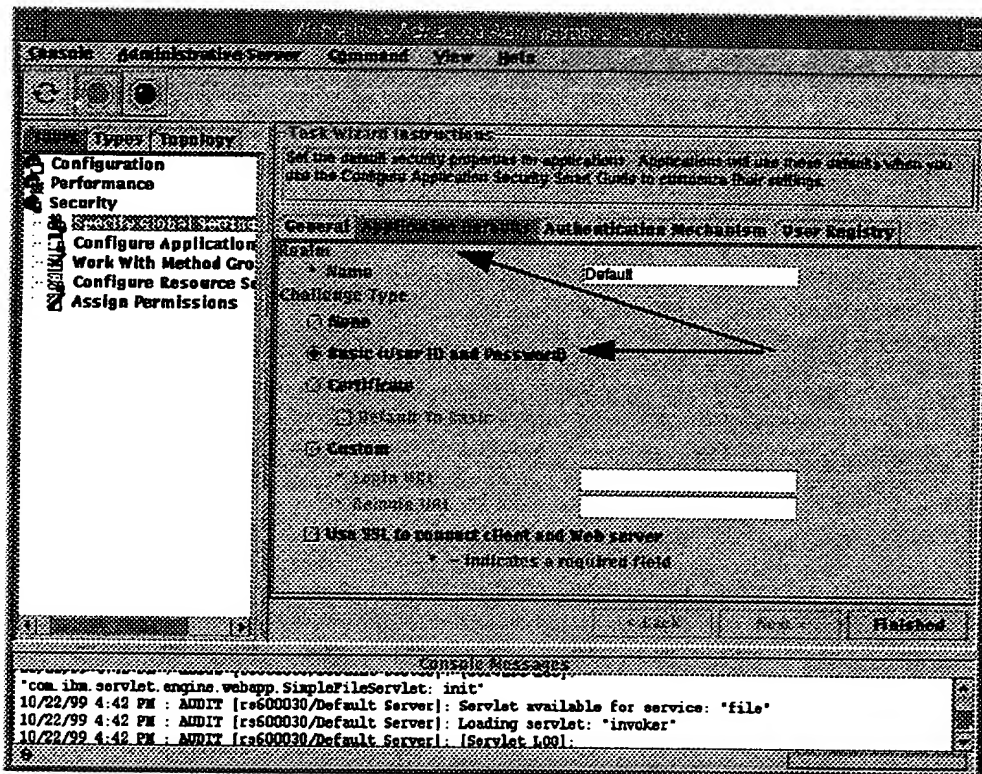
Figure 411. Global settings - Applications Defaults

4. Click the **Authentication Mechanism** tab. On this page you specify the mechanism used by the security server to authenticate the principal's credentials. See Figure 412 on page 431.

Lightweight Third Party Authentication (LTPA): Select this option to use the LDAP directory server as the registry system.

Token Expiration: Specifies how many minutes can pass before a client using an LTPA token must be authenticated again. We used the default value.

Figure 412. Global settings - Authentication Mechanism

5. Click the **User Registry** tab. On this page we specify basic information to use the LDAP Directory Server as shown in Figure 413 on page 433. These are the attributes that you should change:

Security Server ID: Type a valid user ID registered in the LDAP Directory server used by the Application Server V3 Security server. The user ID should have some administrative privileges.

Security Server Password: Type the password for the security server ID user.

Lightweight Third Party Authentication (LTPA): Select this option to use the LDAP directory server as the registry system.

Token Expiration: Specifies how many minutes can pass before a client using an LTPA token must be authenticated again. We used the default value.
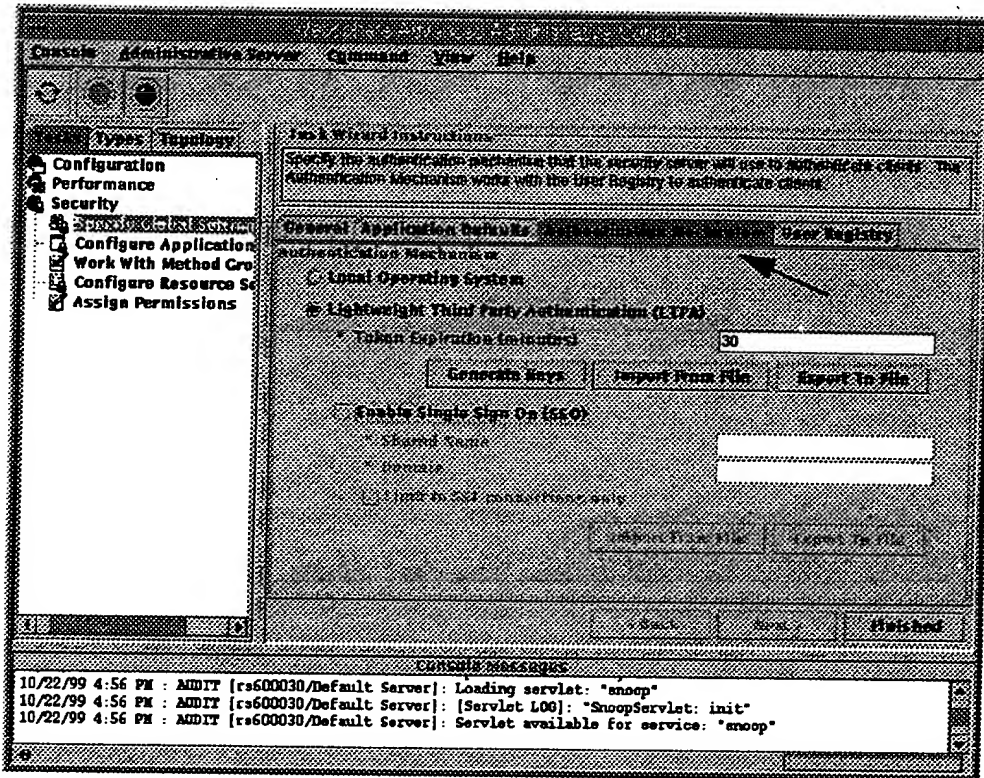

Figure 412. Global settings - Authentication Mechanism

5. Click the **User Registry** tab. On this page we specify basic information to use the LDAP Directory Server as shown in Figure 413 on page 433. These are the attributes that you should change:

Security Server ID: Type a valid user ID registered in the LDAP Directory server used by the Application Server V3 Security server. The user ID should have some administrative privileges.

Security Server Password: Type the password for the security server ID user.

Figure 413. User Registry properties

6. Click the **Finished** button.

   Note: For any change made on the global security settings you must restart the node. To do so, click **Topology -> WebSphere Admin Domain -> Host Name (rs600030)**. Then right-click the mouse button and select **stop** or **restart**. You must leave the administrative console.

You can customize more LDAP attributes such as search filters and certificate mapping as shown in Figure 414:
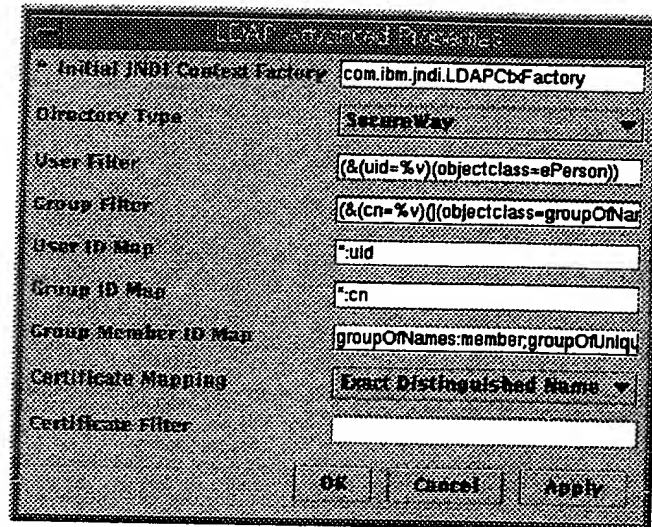
Figure 414. Advanced properties

7. The next step consists of applied security on an application. This means performing the following steps:

   a. Configure an enterprise application.

   b. Configure application security.

   c. Configure resource security.

   d. Assign permissions.

   For more information about how to perform these tasks see the WebSphere documentation available in the directory /usr/WebSphere/AppServer/web/doc/begin_here/index.html

8.1.14 JNDI

JNDI defined by Sun Microsystems provides naming and directory functions to Java programs. JNDI is an API independent of any specific directory service implementation.

The definition prevents, by design, the appearance of any implementation-specific artifacts in the API. The API is designed to cover the common case. JNDI was developed as part of Java Enterprise API set which also includes Enterprise Java beans (EJB) and Java Database Connectivity (JDBC). The EJB specification has a special relationship with JNDI because EJB uses this mechanism to find Entity beans or Sessions beans.

Java™ Media Framework
API Guide

November 19,1999
JMF 2.0 FCS

Contents

Preface

The Java™ Media Framework (JMF) is an application programming interface (API) for incorporating time-based media into Java applications and applets. This guide is intended for Java programmers who want to incorporate time-based media into their applications and for technology providers who are interested in extending JMF and providing JMF plug-ins to support additional media types and perform custom processing and rendering.

About JMF

The JMF 1.0 API (the Java Media Player API) enabled programmers to develop Java programs that presented time-based media. The JMF 2.0 API extends the framework to provide support for capturing and storing media data, controlling the type of processing that is performed during playback, and performing custom processing on media data streams. In addition, JMF 2.0 defines a plug-in API that enables advanced developers and technology providers to more easily customize and extend JMF functionality.

The following classes and interfaces are new in JMF 2.0:

AudioFormat	BitRateControl	Buffer
BufferControl	BufferToImage	BufferTransferHandler
CaptureDevice	CaptureDeviceInfo	CaptureDeviceManager
CloneableDataSource	Codec	ConfigureCompleteEvent
ConnnectionErrorEvent	DataSink	DataSinkErrorEvent
DataSinkEvent	DataSinkListener	Demultiplexer

Effect	EndOfStreamEvent	FileTypeDescriptor
Format	FormatChangeEvent	FormatControl
FrameGrabbingControl	FramePositioningControl	FrameProcessingControl
FrameRateControl	H261Control	H261Format
H263Control	H263Format	ImageToBuffer
IndexedColorFormat	InputSourceStream	KeyFrameControl
MonitorControl	MpegAudioControl	Multiplexer
NoStorageSpaceErrorEvent	PacketSizeControl	PlugIn
PlugInManager	PortControl	Processor
ProcessorModel	PullBufferDataSource	PullBufferStream
PushBufferDataSource	PushBufferStream	QualityControl
Renderer	RGBFormat	SilenceSuppressionControl
StreamWriterControl	Track	TrackControl
VideoFormat	VideoRenderer	YUVFormat

In addition, the MediaPlayer Java Bean has been included with the JMF API in javax.media.bean.playerbean. MediaPlayer can be instantiated directly and used to present one or more media streams.

Future versions of the JMF API will provide additional functionality and enhancements while maintaining compatibility with the current API.

Design Goals for the JMF API

JMF 2.0 supports media capture and addresses the needs of application developers who want additional control over media processing and rendering. It also provides a plug-in architecture that provides direct access to media data and enables JMF to be more easily customized and extended. JMF 2.0 is designed to:

- Be easy to program
- Support capturing media data
- Enable the development of media streaming and conferencing applications in Java

- Enable advanced developers and technology providers to implement custom solutions based on the existing API and easily integrate new features with the existing framework

- Provide access to raw media data

- Enable the development of custom, downloadable demultiplexers, codecs, effects processors, multiplexers, and renderers (JMF *plug-ins*)

- Maintain compatibility with JMF 1.0

About the JMF RTP APIs

The classes in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp` provide support for RTP (Real-Time Transport Protocol). RTP enables the transmission and reception of real-time media streams across the network. RTP can be used for media-on-demand applications as well as interactive services such as Internet telephony.

JMF-compliant implementations are not required to support the RTP APIs in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp`. The reference implementations of JMF provided by Sun Microsystems, Inc. and IBM Corporation fully support these APIs.

The first version of the JMF RTP APIs (referred to as the RTP Session Manager API) enabled developers to receive RTP streams and play them using JMF. In JMF 2.0, the RTP APIs also support the transmission of RTP streams.

The following RTP classes and interfaces are new in JMF 2.0:

SendStream	SendStreamListener	InactiveSendStreamEvent
ActiveSendStreamEvent	SendPayloadChangeEvent	NewSendStreamEvent
GlobalTransmissionStats	TransmissionStats	

The RTP packages have been reorganized and some classes, interfaces, and methods have been renamed to make the API easier to use. The package reorganization consists of the following changes:

- The RTP event classes that were in `javax.media.rtp.session` are now in `javax.media.rtp.event`.

- The RTCP-related classes that were in `javax.media.rtp.session` are now in `javax.media.rtp.rtcp`.

- The rest of the classes in `javax.media.rtp.session` are now in `javax.media.rtp` and the `javax.media.rtp.session` package has been removed.

The name changes consist primarily of the removal of the RTP and RTCP prefixes from class and interface names and the elimination of non-standard abbreviations. For example, `RTPRecvStreamListener` has been renamed to `ReceiveStreamListener`. For a complete list of the changes made to the RTP packages, see the JMF 2.0 Beta release notes.

In addition, changes were made to the RTP APIs to make them compatible with other changes in JMF 2.0:

- `javax.media.rtp.session.io` and `javax.media.rtp.session.depacketizer` have been removed. Custom RTP packetizers and depacketizers are now supported through the JMF 2.0 plug-in architecture. Existing depacketizers will need to be ported to the new plug-in architecture.

- `Buffer` is now the basic unit of transfer between the `SessionManager` and other JMF objects, in place of `DePacketizedUnit` and `DePacketizedObject`. RTP-formatted `Buffers` have a specific format for their data and header objects.

- `BaseEncodingInfo` has been replaced by the generic JMF `Format` object. An RTP-specific `Format` is differentiated from other formats by its encoding string. Encoding strings for RTP-specific `Formats` end in `_RTP`. Dynamic payload information can be provided by associating a dynamic payload number with a `Format` object.

Design Goals for the JMF RTP APIs

The RTP APIs in JMF 2.0 support the reception and transmission of RTP streams and address the needs of application developers who want to use RTP to implement media streaming and conferencing applications. These APIs are designed to:

- Enable the development of media streaming and conferencing applications in Java

- Support media data reception and transmission using RTP and RTCP

- Support custom packetizer and depacketizer plug-ins through the JMF 2.0 plug-in architecture.

- Be easy to program

Partners in the Development of the JMF API

The JMF 2.0 API is being jointly designed by Sun Microsystems, Inc. and IBM Corporation.

The JMF 1.0 API was jointly developed by Sun Microsystems Inc., Intel Corporation, and Silicon Graphics, Inc.

Contact Information

For the latest information about JMF, visit the Sun Microsystems, Inc. website at:

```
http://java.sun.com/products/java-media/jmf/
```

Additional information about JMF can be found on the IBM Corporation website at:

```
http://www.software.ibm.com/net.media/
```

About this Document

This document describes the architecture and use of the JMF 2.0 API. It replaces the Java Media Player Guide distributed in conjunction with the JMF 1.0 releases.

Except where noted, the information in this book is not implementation specific. For examples specific to the JMF reference implementation developed by Sun Microsystems and IBM corporation, see the sample code and solutions available from Sun's JMF website (`http://java.sun.com/products/java-media/jmf/index.html`).

Guide to Contents

This document is split into two parts:

- Part 1 describes the features provided by the JMF 2.0 API and illustrates how you can use JMF to incorporate time-based media in your Java applications and applets.

- Part 2 describes the support for real-time streaming provided by the JMF RTP APIs and illustrates how to send and receive streaming media across the network.

Part 1 is organized into six chapters:

- **"Working with Time-Based Media"**—sets the stage for JMF by introducing the key concepts of media content, presentation, processing, and recording.

- **"Understanding JMF"**—introduces the JMF 2.0 API and describes the high-level architecture of the framework.

- **"Presenting Time-Based Media with JMF"**—describes how to use JMF Players and Processors to present time-based media.

- **"Processing Time-Based Media with JMF"**—describes how to manipulate media data using a JMF Processor.

- **"Capturing Time-Based Media with JMF"**—describes how to record media data using JMF DataSources and Processors.

- **"Extending JMF"**—describes how to enhance JMF functionality by creating new processing plug-ins and implementing custom JMF classes.

Part 2 is organized into six chapters:

- **"Working with Real-Time Media Streams"**—provides an overview of streaming media and the Real-time Transport protocol (RTP).

- **"Understanding the JMF RTP API"**—describes the JMF RTP APIs.

- **"Receiving and Presenting RTP Media Streams"**—illustrates how to handle RTP Client operations.

- **"Transmitting RTP Media Streams"**—illustrates how to handle RTP Server operations.

- **"Importing and Exporting RTP Media Streams"**—shows how to read and write RTP data to a file.

- **"Creating Custom Packetizers and Depacketizers"**—describes how to use JMF plug-ins to support additional RTP packet formats and codecs.

At the end of this document, you'll find Appendices that contain complete sample code for some of the examples used in these chapters and a glossary of JMF-specific terms.

Change History

Version JMF 2.0 FCS

- Fixed references to TrackControl methods to reflect modified TrackControl API.
- Fixed minor sample code errors.
- Clarified behavior of cloneable data sources.
- Clarified order of events when writing to a file.

Version 0.9

Internal Review Draft

Version 0.8

JMF 2.0 Beta draft:

- Added an introduction to RTP, Working with Real-Time Media Streams, and updated the RTP chapters.
- Updated to reflect API changes since the Early Access release.
- Added an example of registering a plug-in with the PlugInManager.
- Added chapter, figure, table, and example numbers and changed the example code style.

Version 0.7

JMF 2.0 Early Access Release 1 draft:

- Updated and expanded RTP chapters in Part 2.
- Added Demultiplexer example to "Extending JMF".
- Updated to reflect API changes since the public review.

Version 0.6

Internal Review Draft

Version 0.5

JMF 2.0 API public review draft.

- Added new concepts chapter, **"Working with Time-Based Media"**.
- Reorganized architecture information in **"Understanding JMF"**.

- Incorporated RTP Guide as Part 2.

Version 0.4

JMF 2.0 API licensee review draft.

Comments

Please submit any comments or suggestions you have for improving this document to jmf-comments@eng.sun.com.

JavaMail™ API
Design Specification

Version 1.1

Send feedback to javamail@sun.com

Contents

Introduction

In the few years since its first release, the Java™ programming language has matured to become a platform. The Java platform has added functionality, including distributed computing with RMI and the CORBA bridge, and a component architecture (JavaBeans™). Java applications have also matured, and many now need an addition to the Java platform: a mail and messaging framework. The JavaMail™ API described in this specification satisfies that need.

The JavaMail API provides a set of abstract classes defining objects that comprise a mail system. The API defines classes like Message, Store and Transport. The API can be extended and can be subclassed to provide new protocols and to add functionality when necessary.

In addition, the API provides concrete subclasses of the abstract classes. These subclasses, including MimeMessage and MimeBodyPart, implement widely used Internet mail protocols and conform to specifications RFC822 and RFC2045. They are ready to be used in application development.

Target Audience

The JavaMail API is designed to serve several audiences:

- Client, server, or middleware developers interested in building mail and messaging applications using the Java programming language.

- Application developers who need to "mail-enable" their applications.

- Service Providers who need to implement specific access and transfer protocols. For example; a telecommunications company can use the JavaMail API to implement a PAGER Transport protocol that sends mail messages to alphanumeric pagers.

Acknowledgments

The authors of this specification are John Mani, Bill Shannon, Max Spivak, Kapono Carter and Chris Cotton.

We would like to acknowledge the following people for their comments and feedback on the initial drafts of this document:

- Terry Cline, John Russo, Bill Yeager and Monica Gaines: Sun Microsystems.

- Arn Perkins and John Ragan: Novell, Inc.

- Nick Shelness: Lotus Development Corporation.

- Juerg von Kaenel: IBM Corporation.

- Prasad Yendluri, Jamie Zawinski, Terry Weissman and Gena Cunanan: Netscape Communications Corporation.

Chapter 2:

Goals and Design Principles

The JavaMail API is designed to make adding electronic mail capability to simple applications easy, while also supporting the creation of sophisticated user interfaces. It includes appropriate convenience classes which encapsulate common mail functions and protocols. It fits with other packages for the Java platform in order to facilitate its use with other Java APIs, and it uses familiar programming models.

The JavaMail API is therefore designed to satisfy the following development and runtime requirements:

- Simple, straightforward class design is easy for a developer to learn and implement.

- Use of familiar concepts and programming models support code development that interfaces well with other Java APIs.

   - Uses familiar exception-handling and JDK 1.1 event-handling programming models.

   - Uses features from the JavaBeans Activation Framework (JAF) to handle access to data based on data-type and to facilitate the addition of data types and commands on those data types. The JavaMail API provides convenience functions to simplify these coding tasks.

- Lightweight classes and interfaces make it easy to add basic mail-handling tasks to any application.

- Supports the development of robust mail-enabled applications, that can handle a variety of complex mail message formats, data types, and access and transport protocols.

The JavaMail API draws heavily from IMAP, MAPI, CMC, c-client and other messaging system APIs: many of the concepts present in these other systems are also present in the JavaMail API. It is simpler to use because it uses features of the Java programming language not available to these other APIs, and because it uses the Java programming language's object model to shelter applications from implementation complexity.

The JavaMail API design is driven by the needs of the applications it supports—but it is also important to consider the needs of API implementors. It is critically important to enable the implementation of messaging systems written using the Java programming language that interoperate with existing messaging systems—especially

Internet mail. It is also important to anticipate the development of new messaging systems. The JavaMail API conforms to current standards while not being so constrained by current standards that it stifles future innovation.

The JavaMail API supports many different messaging system implementations—different message stores, different message formats, and different message transports. The JavaMail API provides a set of base classes and interfaces that define the API for client applications. Many simple applications will only need to interact with the messaging system through these base classes and interfaces.

JavaMail subclasses can expose additional messaging system features. For instance, the MimeMessage subclass exposes and implements common characteristics of an Internet mail message, as defined by RFC822 and MIME standards. Developers can subclass JavaMail classes to provide the implementations of particular messaging systems, such as IMAP4, POP3, and SMTP.

The base JavaMail classes include many convenience APIs that simplify use of the API, but don't add any functionality. The implementation subclasses are not required to implement those convenience methods. The implementation subclasses must implement only the core classes and methods that provide functionality required for the implementation.

Alternately, a messaging system can choose to implement all of the JavaMail API directly, allowing it to take advantage of performance optimizations, perhaps through use of batched protocol requests. The IMAP4 protocol implementation takes advantage of this approach.

The JavaMail API uses the Java programming language to good effect to strike a balance between simplicity and sophistication. Simple tasks are easy, and sophisticated functionality is possible.

Chapter 3:

Architectural Overview

This section describes the JavaMail architecture, defines major classes and interfaces comprising that architecture, and lists major functions that the architecture implements.

JavaMail provides elements that are used to construct an interface to a messaging system, including system components and interfaces. While this specification does not define any specific implementation, JavaMail does include several classes that implement RFC822 and MIME Internet messaging standards. These classes are delivered as part of the JavaMail class package.

JavaMail Layered Architecture

The JavaMail architectural components are layered as shown below:

- The *Abstract Layer* declares classes, interfaces and abstract methods intended to support mail handling functions that all mail systems support. API elements comprising the Abstract Layer are intended to be subclassed and extended as necessary in order to support standard data types, and to interface with message access and message transport protocols as necessary.

- The internet *implementation layer* implements part of the abstract layer using internet standards - RFC822 and MIME.

- JavaMail uses the JavaBeans Activation Framework (JAF) in order to encapsulate message data, and to handle commands intended to interact with that data. Interaction with message data should take place via JAF-aware JavaBeans, which are not provided by the JavaMail API.

JavaMail clients use the JavaMail API and Service Providers implement the JavaMail API. The layered design architecture allows clients to use the same JavaMail API calls to send, receive and store a variety of messages using different data-types from different message stores and using different message transport protocols.

FIGURE 3-1

Mail-enabled Application

```
Java Bean - used to interact and
display message content
```

JavaMail API

JavaMail
Abstract Class Layer

Internet Mail
Implementation Class Layer

IMAP / POP3 / NNTP implementation Layer
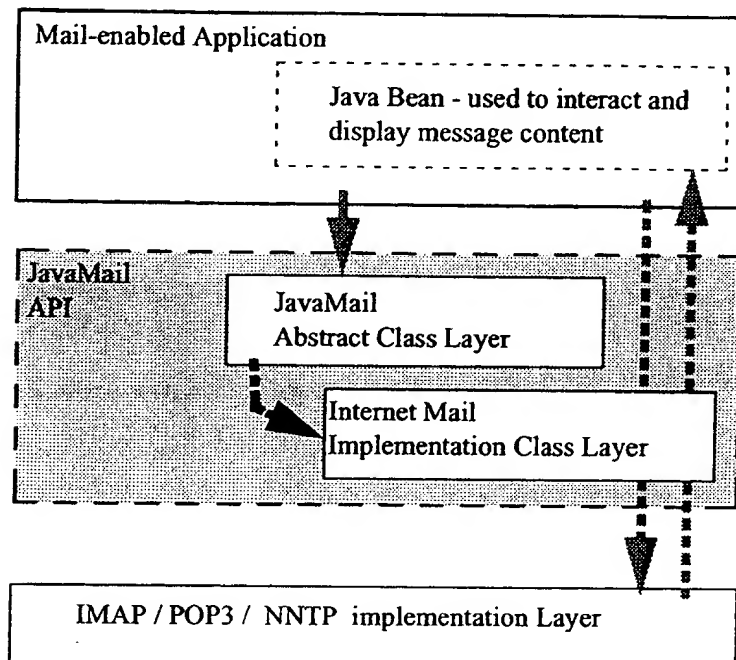
JavaMail Class Hierarchy

The figure below shows major classes and interfaces comprising the JavaMail API. See "Major JavaMail API Components" on page 10 for brief descriptions of all components shown on this diagram.

FIGURE 3-2
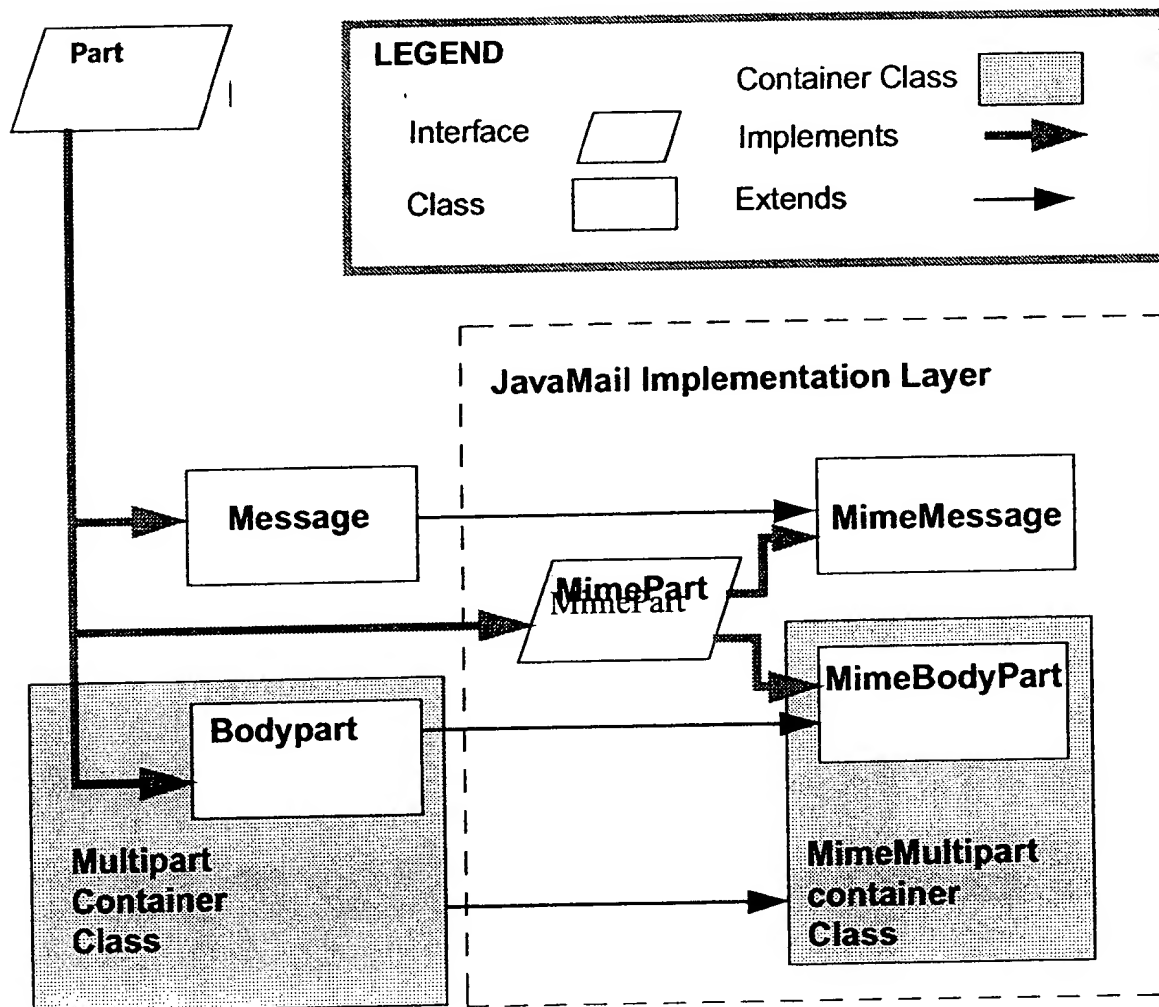
The JavaMail Framework

The JavaMail API is intended to perform the following functions, which comprise the standard mail handling process for a typical client application:

- Create a mail message consisting of a collection of header attributes and a block of data of some known data type as specified in the Content-Type header field. JavaMail uses the Part interface and the Message class to define a mail message. It uses the JAF-defined DataHandler object to contain data placed in the message.

- Create a Session object, which authenticates the user, and controls access to the message store and transport.

- Send the message to its recipient list.

- Retrieve a message from a message store.

- Execute a high-level command on a retrieved message. High-level commands like *view* and *print* are intended to be implemented via JAF-Aware JavaBeans.

Note – The JavaMail framework does not define mechanisms that support message delivery, security, disconnected operation, directory services or filter functionality. Security, disconnected operation and filtering support will be added in future releases.

This figure illustrates the JavaMail message-handling process.

FIGURE 3-3
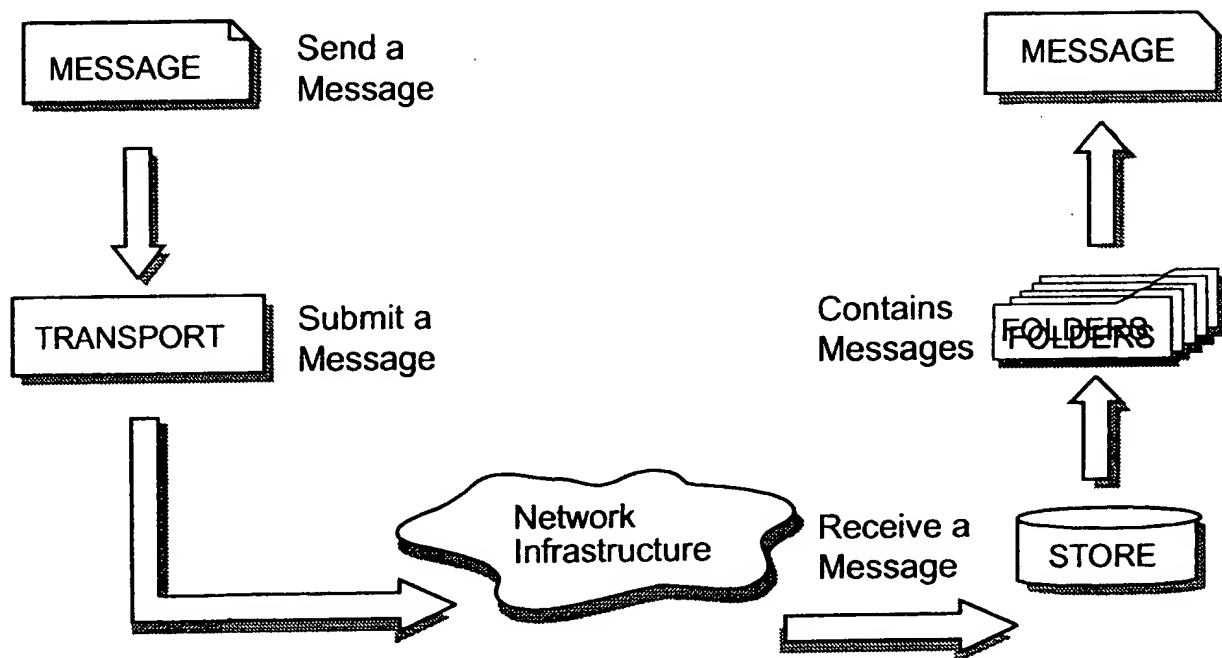
Major JavaMail API Components

This section reviews major components comprising the JavaMail architecture.

The Message Class

The Message class is an abstract class that defines a set of attributes and a content for a mail message. Attributes of the Message class specify addressing information and define the structure of the content, including the content type. The content is represented as a DataHandler object that wraps around the actual data.

The Message class implements the Part interface. The Part interface defines attributes that are required to define and format data content carried by a Message object, and to interface successfully to a mail system. The Message class adds From, To, Subject, Reply-To, and other attributes necessary for message routing via a message transport system. When contained in a folder, a Message object has a set of flags associated with it. JavaMail provides Message subclasses that support specific messaging implementations.

The content of a message is a collection of bytes, or a reference to a collection of bytes, encapsulated within a Message object. JavaMail has no knowledge of the data type or format of the message content. A Message object interacts with its content through an intermediate layer—the JavaBeans Activation Framework (JAF). This separation allows a Message object to handle any arbitrary content and to transmit it using any appropriate transmission protocol by using calls to the same API methods. The message recipient usually knows the content data type and format and knows how to handle that content.

The JavaMail API also supports multipart Message objects, where each Bodypart defines its own set of attributes and content.

Message Storage and Retrieval

Messages are stored in Folder objects. A Folder object can contain subfolders as well as messages, thus providing a tree-like folder hierarchy. The Folder class declares methods that fetch, append, copy and delete messages. A Folder object can also send events to components registered as event listeners.

The Store class defines a database that holds a folder hierarchy together with its messages. The Store class also specifies the *access* protocol that accesses folders and retrieves messages stored in folders. The Store class also provides methods to establish a connection to the database, to fetch folders and to close a connection. Service providers implementing Message Access protocols (IMAP4, POP3 etc.) start off by subclassing the Store class. A user typically starts a session with the mail system by connecting to a particular Store implementation.

Message Composition and Transport

A client creates a new message by instantiating an appropriate Message subclass. It sets attributes like the recipient addresses and the subject, and inserts the content into the Message object. Finally, it sends the Message by invoking the Transport.send method.

The Transport class models the transport agent that routes a message to its destination addresses. This class provides methods that send a message to a list of recipients. Invoking the Transport.send method with a Message object identifies the appropriate transport based on its destination addresses.

The Session Class

The Session class defines global and per-user mail-related properties that define the interface between a mail-enabled client and the network. JavaMail system components use the Session object to set and get specific properties. The Session class also provides a default authenticated session object that desktop applications can share. The Session class is a final concrete class. It cannot be subclassed.

The Session class also acts as a factory for Store and Transport objects that implement specific access and transport protocols. By calling the appropriate factory method on a Session object, the client can obtain Store and Transport objects that support specific protocols.

The JavaMail Event Model

The JavaMail event model conforms to the JDK 1.1 event-model specification, as described in the JavaBeans Specification. The JavaMail API follows the design patterns defined in the JavaBeans Specification for naming events, event methods and event listener registration.

All events are subclassed from the MailEvent class. Clients listen for specific events by registering themselves as listeners for those events. Events notify listeners of state changes as a session progresses. During a session, a JavaMail component generates a specific event-type to notify objects registered as listeners for that event-type. The JavaMail Store, Folder, and Transport classes are event sources. This specification describes each specific event in the section that describes the class that generates that event.

Using the JavaMail API

This section defines the syntax and lists the order in which a client application calls some JavaMail methods in order to access and open a message located in a folder:

1. A JavaMail client typically begins a mail handling task by obtaining the default JavaMail Session object.

```
Session session = Session.getDefaultInstance(
                        props, authenticator);
```

2. The client uses the Session object's getStore method to connect to the default store. The getStore method returns a Store object subclass that supports the access protocol defined in the user properties object, which will typically contain per-user preferences.

```
Store store = session.getStore();
store.connect();
```

3. If the connection is successful, the client can list available folders in the Store, and then fetch and view specific Message objects.

```
// get the INBOX folder
Folder inbox = store.getFolder("INBOX");

// open the INBOX folder
inbox.open(Folder.READ_WRITE);

Message m = inbox.getMessage(1);       // get Message # 1
String subject = m.getSubject();       // get Subject
Object content = m.getContent();       // get content
   . . .
   . . .
```

4. Finally, the client closes all open folders, and then closes the store.

```
inbox.close();            // Close the INBOX
store.close();                 // Close the Store
```

See "Examples Using the JavaMail API" on page 63 for a more complete example.

Chapter 4:
The Message *Class*

The Message class defines a set of attributes and a content for a mail message. Message attributes specify message addressing information and define the structure of the content, including the content type. The content is represented by a DataHandler object that wraps around the actual data. The Message class is an abstract class that implements the Part interface.

Subclasses of the Message classes can implement several standard message formats. For example, the JavaMail API provides the MimeMessage class, that extends the Message class to implement the RFC822 and MIME standards. Implementations can typically construct themselves from byte streams and generate byte streams for transmission.

A Message subclass instantiates an object that holds message content, together with attributes that specify addresses for the sender and recipients, structural information about the message, and the content type of the message body. Messages placed into a folder also have a set of flags that describe the state of the message within the folder.

The figure below illustrates the structure of the Message class.

FIGURE 4-1
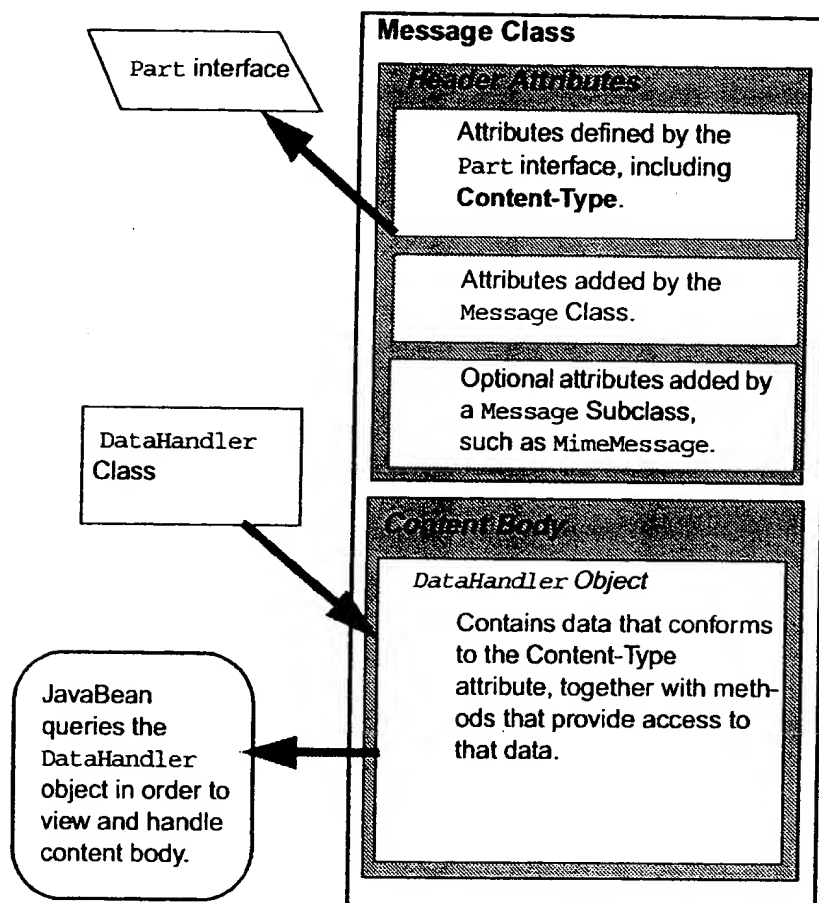
The Message object has no direct knowledge of the nature or semantics of its content. This separation of structure from content allows the message object to contain any arbitrary content.

Message objects are either retrieved from a Folder object or constructed by instantiating a new Message object of the appropriate subclass. Messages stored within a Folder object are sequentially numbered, starting at one. An assigned message number can change when the folder is expunged, since the expunge operation removes deleted messages from the folder and also renumbers the remaining messages.

A Message object can contain multiple parts, where each part contains its own set of attributes and content. The content of a multipart message is a Multipart object that contains BodyPart objects representing each individual part. The Part interface defines the structural and semantic similarity between the Message class and the BodyPart class.

The figure below illustrates a Message instance hierarchy, where the message contains attributes, a set of flags, and content. See "MimeMessage Object Hierarchy" on page 81 for an illustration of the MimeMessage object hierarchy.

FIGURE 4-2
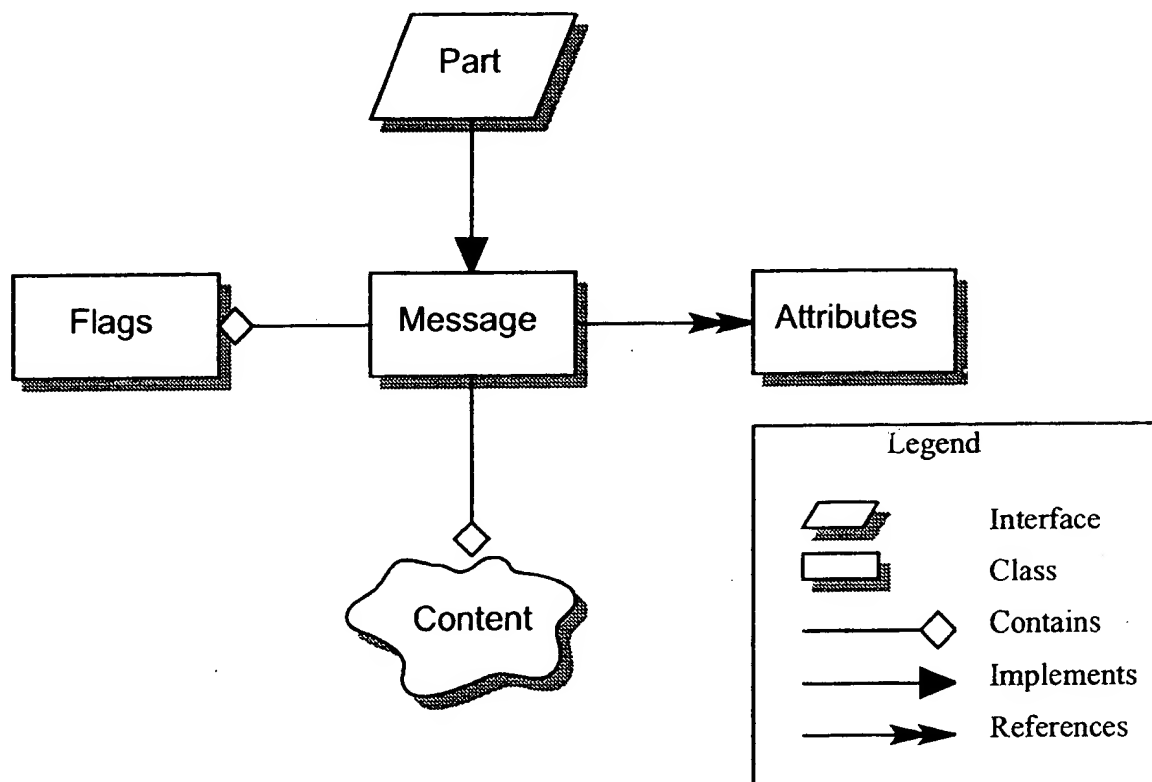
The Message class provides methods to perform the following tasks:

■ Get, set and create its attributes and content:

```
public String getSubject() throws MessagingException;

public void setSubject(String subject)
    throws MessagingException;

public String[] getHeader(String name)
    throws MessagingException;
```

```
       public void setHeader(String name, String value)
           throws MessagingException;

       public Object getContent()
           throws MessagingException;

       public void setContent(Object content, String type)
           throws MessagingException
```

- Save changes to its containing folder.

```
       public void saveChanges()
           throws MessagingException;
```

This method also ensures that the Message header fields are updated to be consistent with the changed message contents.

- Generate a bytestream for the Message object.

```
       public void writeTo(OutputStream os)
           throws IOException, MessagingException;
```

This byte stream can be used to save the message or send it to a Transport object.

The Part Interface

The Part interface defines a set of standard headers common to most mail systems, specifies the data-type assigned to data comprising a content block, and defines set and get methods for each of these members. It is the basic data component in the JavaMail API and provides a common interface for both the Message and BodyPart classes. See the JavaMail API (Javadoc) documentation for details.

Note – A Message object can not be contained directly in a Multipart object, but must be embedded in a BodyPart first.

Message Attributes

The Message class adds its own set of standard attributes to those it inherits from the Part interface. These attributes include the sender and recipient addresses, the subject, flags, and sent and received dates. The Message class also supports non-standard attributes in the form of headers. See the JavaMail API (Javadoc) Documentation for the list of standard attributes defined in the Message class. Not all messaging systems will support arbitrary headers, and the availability and meaning of particular header names is specific to the messaging system implementation.

The ContentType *Attribute*

The contentType attribute specifies the data type of the content, following the MIME typing specification (RFC 2045). A MIME type is composed of a primary type that declares the general type of the content, and a subtype that specifies a specific format for the content. A MIME type also includes an optional set of type-specific parameters.

JavaMail API components can access content via these mechanisms:

As an input stream	The Part interface declares the getInputStream method that returns an input stream to the content. Note that Part implementations must decode any mail-specific transfer encoding before providing the input stream.
As a DataHandler object	The Part interface declares the getDataHandler method that returns a javax.activation.DataHandler object that wraps around the content. The DataHandler object allows clients to discover the operations available to perform on the content, and to instantiate the appropriate component to perform those operations. See "The JavaBeans Activation Framework" on page 41 for details describing the data typing framework
As an object in the Java programming language	The Part interface declares the getContent method that returns the content as an object in the Java programming language. The type of the returned object is dependent on the content's data type. If the content is of type multipart, the getContent method returns a Multipart object, or a Multipart subclass object. The getContent method returns an input stream for unknown content-types. Note that the getContent method uses the DataHandler internally to obtain the native form.

The setDataHandler(DataHandler) method specifies content for a new Part object, as a step toward the construction of a new message. The Part also provides some convenience methods to set up most common content types.

Part provides the writeTo method that writes its byte stream in mail-safe form suitable for transmission. This byte stream is typically an aggregation of the Part attributes and the byte stream for its content.

The Address *Class*

The Address class represents email addresses. The Address class is an abstract class. Subclasses provide implementation-specific semantics.

The BodyPart *Class*

The BodyPart class is an abstract class that implements the Part interface in order to define the attribute and content body definitions that Part declares. It does not declare attributes that set From, To, Subject, ReplyTo, or other address header fields, as a Message object does.

A BodyPart object is intended to be inserted into a Multipart container, later accessed via a multipart message.

The Multipart Class

The Multipart class implements multipart messages. A multipart message is a Message object where the content-type specifier has been set to *multipart*. The Multipart class is a container class that contains objects of type Bodypart. A Bodypart object is an instantiation of the Part interface—it contains either a new Multipart container object, or a DataHandler object.

The figure below illustrates the structure and content of a multipart message:

FIGURE 4-3

Message

Header Attributes

Normal Message, includes a **Content-Type attribute** set to 'Multipart.'.

Content Body

Normal Message, includes a Content body of type 'Multipart.'

Multipart Object

Bodypart Object

Header Attributes

Attributes defined by the Part interface only.

Attributes include a second **Content-Type** attribute.

Content Body

The content body itself can be **either** a DataHandler object containing data, **or** another Multipart object.

Bodypart Object

A Multipart Message can hold more than one Bodypart object.

A **multipart message** is a simple message object where the Content-Type is set to 'multipart,' and the Content Body carries a reference to a Multipart object.

A Multipart object is a container of Bodypart objects, where each Bodypart can contain either a DataHandler object, or another Multipart object.

Note that Multipart objects can be nested to any reasonable depth within a multipart message, in order to build an appropriate structure for data carried in DataHandler objects. Therefore, it is important to check the ContentType header for each BodyPart element stored within a Multipart container. The figure below illustrates a typical nested Multipart message.

FIGURE 4-4
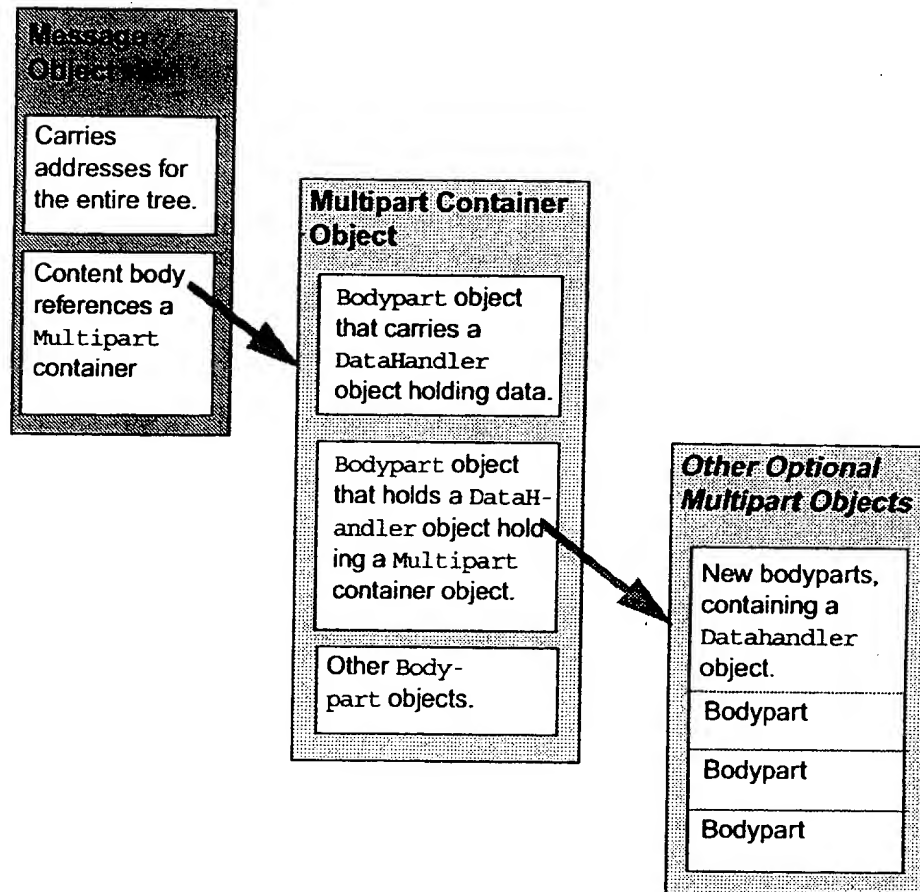
Typically, the client calls the getContentType method to get the content type of a message. If getContentType returns a MIME-type whose primary type is multipart, then the client calls getContent to get the Multipart container object.

The Multipart object supports several methods that get, create, and remove individual BodyPart objects.

```
public int getCount() throws MessagingException;

public Body getBodyPart(int index)
    throws MessagingException;
```

```
public void addBodyPart(BodyPart part)
    throws MessagingException;

public void removeBodyPart(BodyPart body)
    throws MessagingException;

public void removeBodyPart(int index)
    throws MessagingException;
```

The Multipart class implements the javax.beans.DataSource interface. It can act as the DataSource object for javax.beans.DataHandler and javax.beans.DataContentHandler objects. This allows message-aware content handlers to handle multipart data sources more efficiently, since the data has already been parsed into individual parts.

This diagram illustrates the structure of a multipart message, and shows calls from the associated Message and Multipart objects, for a typical call sequence returning a BodyPart containing text/plain content.

FIGURE 4-5
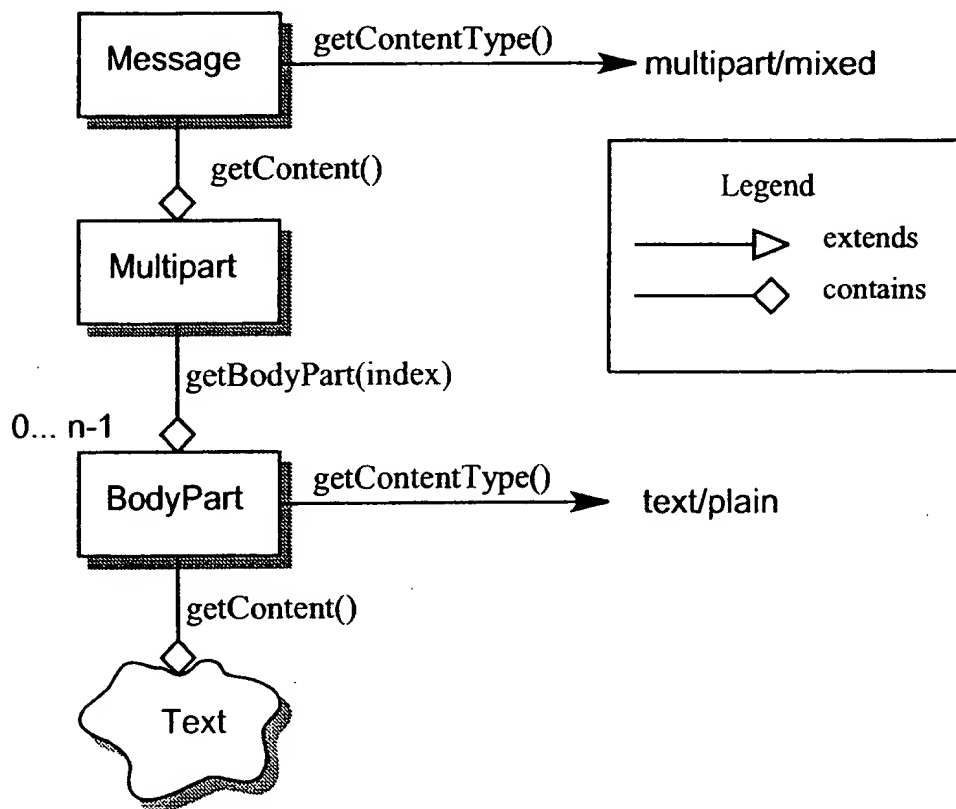
In this figure, the ContentType attribute of a Message object indicates that it holds a multipart content. Use the getContent method to obtain the Multipart object.

The JavaBeans Activation Framework

JavaMail relies heavily on the JavaBeans Activation Framework (JAF) to determine the MIME data type, to determine the commands available on that data, and to provide a software component corresponding to a particular behavior. The JAF specification is part of the "Glasgow" JavaBeans specification. More details can be obtained from http://java.sun.com/beans/glasgow/jaf.html

This section explains how the JavaMail and JAF APIs work together to manage message content. It describes how clients using JavaMail can access and operate on the content of Messages and BodyParts. This discussion assumes you are familiar with the JAF specification posted at http://java.sun.com.

Accessing the Content

For a client using JavaMail, arbitrary data is introduced to the system in the form of mail messages. The javax.mail.Part interface allows the client to access the content. Part consists of a set of attributes and a "content". The Part interface is the common base interface for Messages and BodyParts. A typical mail message has one or more body parts, each of a particular MIME type.

Anything that deals with the content of a Part will use the Part's DataHandler. The content is available through the DataHandlers either as an InputStream or as an object in the Java programming language. The Part also defines convenience methods that call through to the DataHandler. For example: the Part.getContent method is the same as calling Part.getDataHandler().getContent() and the Part.getInputStream method is the same as Part.getDataHandler().getInputStream().

The content returned (either via an InputStream or an object in the Java programmin language) depends on the MIME type. For example: a Part that contains textual content returns the following:

- The Part.getContentType method returns text/plain

- The Part.getInputStream method returns an InputStream containing the bytes of the text

- The Part.getContent method returns a java.lang.String object

Content is returned either as an input stream, or as an object in the Java programming language.

* When an InputStream is returned, any mail-specific encodings are decoded before the stream is returned.

* When an object in the Java programming language is returned using the getContent method, the type of the returned object depends upon the content itself. In the JavaMail API, any Part with a main content type set to "multipart/" (any kind of multipart) should return a javax.mail.Multipart object from the getContent method. A Part with a content type of message/rfc822 returns a javax.mail.Message object from the getContent method.

Example: Message Output

This example shows how you can traverse Parts and display the data contained in a message.

```
public void printParts(Part p) {
    Object o = p.getContent();
    if (o instanceof String) {
        System.out.println("This is a String");
        System.out.println((String)o);
    } else if (o instanceof Multipart) {
        System.out.println("This is a Multipart");
        Multipart mp = (Multipart)o;
        int count = mp.getCount();
        for (int i = 0; i < count; i++) {
            printParts(mp.getBodyPart(i));
        }
    } else if (o instanceof InputStream) {
        System.out.println("This is just an input stream");
        InputStream is = (InputStream)o;
        int c;
        while ((c = is.read()) != -1)
            System.out.write(c);
    }
}
```

Operating on the Content

The DataHandler allows clients to discover the operations available on the content of a Message, and to instantiate the appropriate JavaBeans to perform those operations. The most common operations on Message content are *view, edit* and *print*.

Example: Viewing a Message

Consider a Message "Viewer" Bean that presents a user interface that displays a mail message. This example shows how a viewer bean can be used to display the content of a message (that usually is text/plain, text/html, or multipart/mixed).

Note – Perform error checking to ensure that a valid Component was created.

```
// message passed in as parameter
void setMessage(Message msg) {
    DataHandler dh = msg.getDataHandler();
    CommandInfo cinfo = dh.getCommand("view");
    Component comp = dh.getBean(cinfo);
    this.setMainViewer(comp);
}
```

Example: Showing Attachments

In this example, the user has selected an attachment and wishes to display it in a separate dialog. The client locates the correct viewer object as follows.

```
// Retrieve the BodyPart from the current attachment
BodyPart bp = getSelectedAttachment();

DataHandler dh = bp.getDataHandler();
CommandInfo cinfo = dh.getCommand("view");
Component comp = dh.getBean(cinfo);

// Add viewer to dialog Panel
MyDialog myDialog = new MyDialog();
myDialog.add(viewer);

// display dialog on screen
myDialog.show();
```

See "Setting Message Content" on page 47 for examples that construct a message for a send operation.

Adding Support for Content Types

Support for commands acting on message data is an implementation task left to the client. JavaMail and JAF APIs intend for this support to be provided by a JAF-Aware JavaBean. Almost all data will require edit and view support.

Currently, the JavaMail API does not provide viewer JavaBeans. The JAF does provide two very simple JAF-aware viewer beans: A Text Viewer and Image Viewer. These beans handle data where content-type has been set to text/plain or image/gif.

Developers writing a JavaMail client need to write additional viewers that support some of the basic content types—specifically message/rfc822, multipart/mixed, and text/plain. These are the usual content-types encountered when displaying a Message, and they provide the look and feel of the application.

Content developers providing additional data types should refer to the JAF specification, that discusses how to create DataContentHandlers and Beans that operate on those contents.

Chapter 8:
Message Composition

This section describes the process used to instantiate a message object, add content to that message, and send it to its intended list of recipients.

The JavaMail API allows a client program to create a message of arbitrary complexity. Messages are instantiated from the Message subclass. The client program can manipulate any message as if it had been retrieved from a Store.

Building a Message Object

To create a message, a client program instantiates a Message object, sets appropriate attributes, and then inserts the content.

* The attributes specify the message address and other values necessary to send, route, receive, decode and store the message. Attributes also specify the message structure and data content type.

* Message content is carried in a DataHandler object, that carries either data or a Multipart object. A DataHandler carries the content body and provides methods the client uses to handle the content. A Multipart object is a container that contains one or more Bodypart objects, each of which can in turn contain DataHandler objects.

Message Creation

javax.mail.Message is an abstract class that implements the Part interface. Therefore, to create a message object, select a message subclass that implements the appropriate message type.

For example, to create a Mime message, a JavaMail client instantiates an empty javax.mail.internet.MimeMessage object passing the current Session object to it:

Message msg = new MimeMessage(session);

Setting Message Attributes

The Message class provides a set of methods that specify standard attributes common to all messages. The MimeMessage class provides additional methods that set MIME-specific attributes. The client program can also set non-standard attributes (custom headers) as name-value pairs.

The methods for setting standard attributes are listed below:

```
public class Message {
    public void setFrom(Address addr);
    public void setFrom();    // retrieves from system
    public void setRecipients(RecipientType type, Address[] addrs);
    public void setReplyTo(Address[] addrs);
    public void setSentDate(Date date);
    public void setSubject(String subject);
    ...
}
```

The Part interface specifies the following method, that sets custom headers:

```
public void setHeader(String name, String value)
```

The setRecipients method takes a RecipientType as its first parameter, which specifies which recipient field to use. Currently, Message.RecipientType.TO, Message.RecipientType.CC, and Message.RecipientType.BCC are defined. Additional RecipientTypes may be defined as necessary.

The Message class provides two versions of the of the setFrom method:

- setFrom(Address addr) specifies the sender explicitly from an Address object parameter.

- setFrom retrieves the sender's username from the local system.

The code sample below sets attributes for the MimeMessage just created. First, it instantiates Address objects to be used as To and From addresses. Then, it calls set methods, which equate those addresses to appropriate message attributes:

```
toAddrs[] = new InternetAddress[1];
toAddrs[0] = new InternetAddress("luke@rebellion.gov");
Address fromAddr =
       new InternetAddress("han.solo@smuggler.com");

msg.setFrom(fromAddr);
msg.setRecipients(Message.RecipientType.TO, toAddrs);
msg.setSubject("Takeoff time.");
msg.setSentDate(new Date());
```

Setting Message Content

The Message object carries content data within a DataHandler object. To add
content to a Message, a client creates content, instantiates a DataHandler object,
places content into that DataHandler object, and places that object into a Message
object that has had its attributes defined.

The JavaMail API provides two techniques that set message content. The first
technique uses the setDataHandler method. The second technique uses the
setContent method.

Typically, clients add content to a DataHandler object by calling
setDataHandler(DataHandler) on a Message object. The DataHandler is an
object that encapsulates data. The data is passed to the DataHandler constructor as
either a DataSource (a stream connected to the data) or as an object in the Java
programming language. The InputStream object creates the DataSource. See "The
JavaBeans Activation Framework" on page 41 for additional information.

```
public class DataHandler {
    DataHandler(DataSource dataSource);
    DataHandler(Object data, String mimeType);
}
```

The code sample below shows how to place text content into an InternetMessage.
First, create the text as a string object. Then, pass the string into a DataHandler
object, together with its MIME type. Finally, add the DataHandler object to the
message object:

```
// create brief message text
String content = "Leave at 300.";

// instantiate the DataHandler object

DataHandler data = new DataHandler(content, "text/plain");

// Use setDataHandler() to insert data into the
// new Message object

 msg.setDataHandler(data);
```

Alternately, setContent implements a simpler technique that takes the data object
and its MIME type. setContent creates the DataHandler object automatically:

```
// create the message text
String content = "Leave at 300.";

// call setContent to pass content and content type
// together into the message object

msg.setContent(content, "text/plain");
```

When the client calls Transport.send() to send this message, the recipient will
receive the message below, using either technique:

```
Date: Wed, 23 Apr 1997 22:38:07 -0700 (PDT)
From: han.solo@smuggler.com
Subject: Takeoff time
To: luke@rebellion.gov

Leave at 300.
```

Building a MIME Multipart Message

Follow these steps to create a MIME Multipart Message:

1. Instantiate a new MimeMultipart object, or a subclass.

2. Create MimeBodyParts for the specific message parts. Use the setContent
   method or the setDataHandler method to create the content for each
   Bodypart, as described in the previous section.

Note – The default subtype for a MimeMultipart object is *mixed*. It can be set to other
subtypes as required. MimeMultipart subclasses might already have their subtype
set appropriately.

---

3. Insert the Multipart object into the Message object by calling
   setContent(Multipart) within a newly-constructed Message object.

   The example below creates a Multipart object and then adds two message parts
   to it. The first message part is a text string, "Spaceport Map," and the second
   contains a document of type "application/postscript." Finally, this multipart
   object is added to a MimeMessage object of the type described above.

```
// Instantiate a Multipart object
MimeMultipart mp = new MimeMultipart();

// create the first bodypart object
MimeBodyPart b1 = new MimeBodyPart();

// create textual content
// and add it to the bodypart object
b1.setContent("Spaceport Map","text/plain");
mp.addBodyPart(b1);

// Multipart messages usually have more than
// one body part.  Create a second body part
// object, add new text to it, and place it
// into the multipart message as well. This
// second object holds postscript data.

MimeBodyPart b2 = new MimeBodyPart(); b2.setContent(map,"application/
postscript");
mp.addBodyPart(b2);


// Create a new message object as described above,
// and set its attributes.  Add the multipart
// object to this message and call saveChanges()
// to write other message headers automatically.

Message msg = new MimeMessage(session);

// Set message attrubutes as in a singlepart
// message.

msg.setContent(mp);              // add Multipart
msg.saveChanges();               // save changes
```

After all message parts are created and inserted, call the saveChanges methodto
ensure that the client writes appropriate message headers. This is identical to the
process followed with a single part message. Note that the JavaMail API calls the
saveChanges method implicitly during the send process, so invoking it is
unnecessary and expensive if the message is to be sent immediately.

Using The Transport Class

The code segment below sends a `MimeMessage` using a `Transport` class implementing the SMTP protocol. The client creates two `InternetAddress` objects that specify the recipients and retrieves a `Transport` object from the default `Session` that supports sending messages to Internet addresses. Then the `Session` object uses a `Transport` object to send the message.

```
// Get a session
Session session = Session.getInstance(props, null);

// Create an empty MimeMessage and its part
Message msg = new MimeMessage(session);
... add headers and message parts as before

// create two destination addresses
Address[] addrs = {new InternetAddress("mickey@disney.com"),
                   new InternetAddress("goofy@disney.com")};

// get a transport that can handle sending message to
// InternetAddresses. This will probably map to a transport
// that supports SMTP.
Transport trans = session.getTransport(addrs[0]);

// add ourselves as ConnectionEvent and TransportEvent listeners
trans.addConnectionListener(this);
trans.addTransportListener(this);

// connect method determines what host to use from the
// session properties
trans.connect();

// send the message to the addresses we specified above
trans.sendMessage(msg, addrs);
```

Chapter 10:

Internet Mail

The JavaMail specification does not define any implementation. However, the API does include a set of classes that implement Internet Mail standards. Although not part of the specification, these classes can be considered part of the JavaMail package. They show how to adapt an existing messaging architecture to the JavaMail framework.

These classes implement the Internet Mail Standards defined by the RFCs listed below:

- RFC822 (Standard for the Format of Internet Text Messages)
- RFC2045, RFC2046, RFC2047 (MIME)

RFC822 describes the structure of messages exchanged across the Internet. Messages are viewed as having a header and contents. The header is composed of a set of standard and optional header fields. The header is separated from the content by a blank line. The RFC specifies the syntax for all header fields and the semantics of the standard header fields. It does not however, impose any structure on the message contents.

The MIME RFCs 2045, 2046 and 2047 define message content structure by defining structured body parts, a typing mechanism for identifying different media types, and a set of encoding schemes to encode data into mail-safe characters.

The Internet Mail package allows clients to create, use and send messages conforming to the standards listed above. It gives service providers a set of base classes and utilities they can use to implement Stores and Transports that use the Internet mail protocols. See "MimeMessage Object Hierarchy" on page 81 for a Mime class and interface hierarchy diagram.

The JavaMail MimePart interface models an *entity* as defined in RFC2045, Section 2.4. MimePart extends the JavaMail Part interface to add MIME-specific methods and semantics. The MimeMessage and MimeBodyPart classes implement the MimePart interface. The following figure shows the class hierarchy of these classes.

FIGURE 10-1
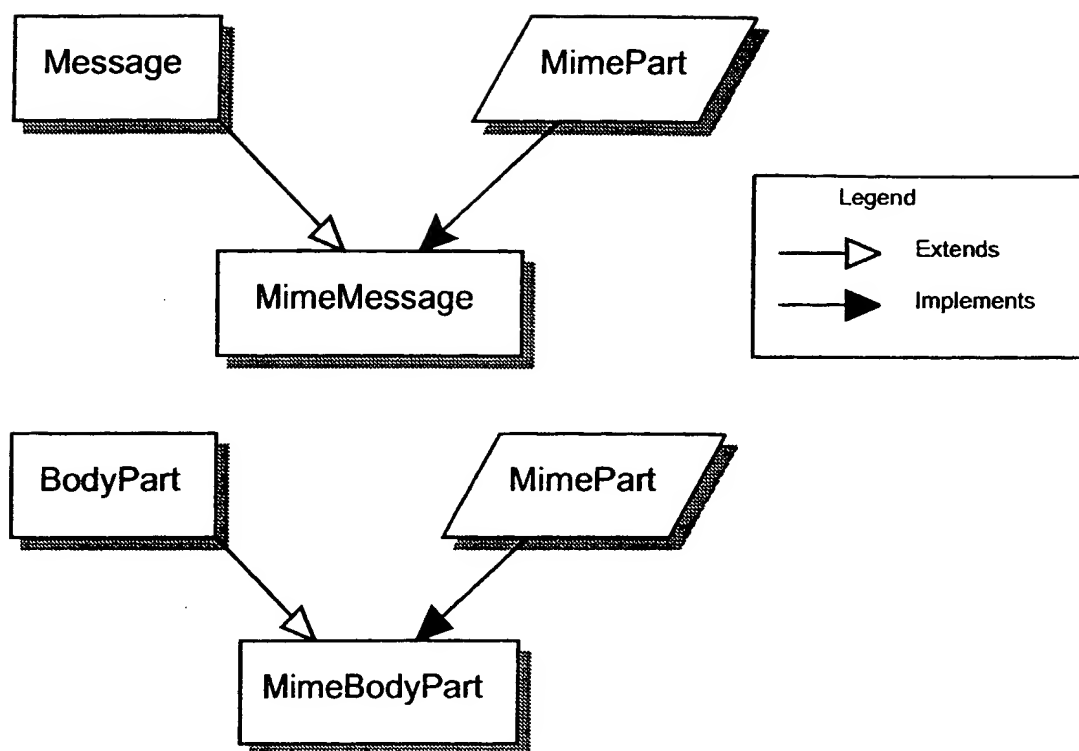
The MimeMessage Class

The MimeMessage class extends Message and implements MimePart. This class implements an email message that conforms to the RFC822 and MIME standards.

The MimeMessage class provides a default constructor that creates an empty MimeMessage object. The client can fill in the message later by invoking the parse method on an RFC822 input stream. Note that the parse method is protected, so that only this class and its subclasses can use this method. Service providers implementing 'light-weight' Message objects that are filled in on demand can generate the appropriate byte stream and invoke the parse method when a component is requested from a message. Service providers that can provide a separate byte stream for the message body (distinct from the message header) can override the getContentStream method.

The client can also use the default constructor to create new MimeMessage objects for sending. The client sets appropriate attributes and headers, inserts content into the message object, and finally calls the send method for that MimeMessage object.

This code sample creates a new `MimeMessage` object for sending. See "Message Composition" on page 45 and "Transport Protocols and Mechanisms" on page 51 for details.

```
MimeMessage m = new MimeMessage(session);
// Set FROM:
m.setFrom(new InternetAddress("jmk@Sun.COM"));
// Set TO:
InternetAddress a[] = new InternetAddress[1];
a[0] = new InternetAddress("javamail@Sun.COM");
m.setRecipients(Message.RecipientType.TO, a);
// Set content
m.setContent(data, "text/plain");
// Send message
m.send();
```

The `MimeMessage` class also provides a constructor that uses an input stream to instantiate itself. The constructor internally invokes the parse method to fill in the message. The `InputStream` object is left positioned at the end of the message body.

```
InputStream in = getMailSource(); // a stream of mail messages
MimeMessage m = null;
for (; ;) {
    try {
        m = new MimeMessage(session, in);
    } catch (MessagingException ex) {
        // reached end of message stream
        break;
    }
}
```

`MimeMessage` implements the `writeTo` method by writing an RFC822-formatted byte stream of its headers and body. This is accomplished in two steps: First, the `MimeMessage` object writes out its headers; then it delegates the rest to the `DataHandler` object representing the content.

The `MimeBodyPart` Class

The `MimeBodyPart` class extends `BodyPart` and implements the `MimePart` interface. This class represents a `Part` inside a `Multipart`. `MimeBodyPart` implements a `BodyPart` as defined by RFC2045, Section 2.5.

The `getBodyPart(int index)` returns the `MimeBodyPart` object at the given index. `MimeMultipart` also allows the client to fetch `MimeBodyPart` objects based on their Content-IDs.

The `addBodyPart` method adds a new `MimeBodyPart` object to a `MimeMultipart` as a step towards constructing a new multipart `MimeMessage`.

The MimeMultipart *Class*

The MimeMultipart class extends Multipart and models a MIME multipart content within a message or a body part.

A MimeMultipart is obtained from a MimePart containing a ContentType attribute set to multipart, by invoking that part's getContent method.

The client creates a new MimeMultipart object by invoking its default constructor. To create a new multipart MimeMessage, create a MimeMultipart object (or its subclass); use set methods to fill in the appropriate MimeBodyParts; and finally, use setContent(Multipart) to insert it into the MimeMessage.

MimeMultipart also provides a constructor that takes an input stream positioned at the beginning of a MIME multipart stream. This class parses the input stream and creates the child body parts.

The getSubType method returns the multipart message MIME subtype. The subtype defines the relationship among the individual body parts of a multipart message. More semantically complex multipart subtypes are implemented as subclasses of MimeMultipart, providing additional methods that expose specific functionality.

Note that a multipart content object is treated like any other content. When parsing a MIME Multipart stream, the JavaMail implementation uses the JAF framework to locate a suitable DataContentHandler for the specific subtype and uses that handler to create the appropriate Multipart instance. Similarly, when generating the output stream for a Multipart object, the appropriate DataContentHandler is used to generate the stream.

The MimeUtility *Class*

MimeUtility is a utility class that provides MIME-related functions. All methods in this class are static methods. These methods currently perform the functions listed below:

Content Encoding and Decoding

Data sent over RFC 821/822-based mail systems are restricted to 7-bit US-ASCII bytes. Therefore, any non-US-ASCII content needs to be encoded into the 7-bit US-ASCII (mail-safe) format. MIME (RFC 2045) specifies the "base64" and "quoted-printable" encoding schemes to perform this encoding. The following methods support content encoding:

- The getEncoding method takes a DataSource object and returns the Content-Transfer-Encoding that should be applied to the data in that DataSource object to make it mail-safe.

- The encode method wraps an encoder around the given output stream based on the specified Content-Transfer-Encoding. The decode method decodes the given input stream, based on the specified Content-Transfer-Encoding.

Header Encoding and Decoding

RFC 822 restricts the data in message headers to 7bit US-ASCII characters. MIME (RFC 2047) specifies a mechanism to encode non 7bit US-ASCII characters so that they are suitable for inclusion in message headers. This section describes the methods that enable this functionality.

The header-related methods (getHeader, setHeader) in Part and Message operate on Strings. String objects contain (16 bit) Unicode characters.

Since RFC 822 prohibits non US-ASCII characters in headers, clients invoking the setHeader() methods must ensure that the header values are appropriately encoded if they contain non US-ASCII characters.

The encoding process (based on RFC 2047) consists of two steps:

1. Convert the Unicode String into an array of bytes in another charset. This step is required because Unicode is not yet a widely used charset. Therefore, a client must convert the Unicode characters into a charset that is more palatable to the recipient.

2. Apply a suitable encoding format that ensures that the bytes obtained in the previous step are mail-safe.

The encodeText method combines the two steps listed above to create an encoded header. Note that as RFC 2047 specifies, only "unstructured" headers and user-defined extension headers can be encoded. It is prudent coding practice to run such header values through the encoder to be safe. Also note that the encodeText method encodes header values only if they contain non US-ASCII characters.

The reverse of this process (decoding) needs to be performed when handling header values obtained from a MimeMessage or MimeBodyPart using the getHeader set of methods, since those headers might be encoded as per RFC 2047. The decodeText method takes a header value, applies RFC 2047 decoding standards, and returns the

decoded value as a Unicode String. Note that this method should be invoked only on "unstructured" or user-defined headers. Also note that decodeText attempts decoding only if the header value was encoded in RFC 2047 style. It is advised that you always run header values through the decoder to be safe.

The ContentType *Class*

The ContentType class is a utility class that parses and generates MIME content-type headers.

To parse a MIME content-Type value, create a ContentType object and invoke the toString method.

The ContentType class also provides methods that match Content-Type values.

The following code fragment illustrates the use of this class to extract a MIME parameter.

```
String type = part.getContentType();
ContentType cType = new ContentType(type);

if (cType.match("application/x-foobar"))
    iString color = cType.getParameter("color");
```

This code sample uses this class to construct a MIME Content-Type value:

```
ContentType cType = new ContentType();
cType.setPrimaryType("application");
cType.setSubType("x-foobar");
cType.setParameter("color", "red");

String contentType = cType.toString();
```

JavaMail API Design Specification

Please Recycle

Adobe PostScript

August 1998

1.   (Original) A method of email administration,

the method implemented in a transcoding gateway, the transcoding gateway comprising client device records stored in computer memory, each client device record representing a client device, each client device record including

a mailbox address field,

an internet address field,

a digital file format code field, and

a path name field,

the transcoding gateway further comprising at least one file system, each file system further comprising file system storage locations, each file system storage location having a path name,

the method comprising the steps of:

receiving in the transcoding gateway an email message, the email message comprising at least one destination mailbox address, the email message further comprising at least one digital object;

transcoding the digital object into a digital file having a digital format and a file name; and

downloading the digital file to a destination client device at an internet address recorded in an internet address field of a client device record, the client device record having:

recorded in the client device record's mailbox address field, a mailbox address identical to the destination mailbox address of the email message, and,

recorded in the client device record's digital file format code field, a digital file format code indicating that the client device represented by the client device record is capable of receiving the digital format of the digital file.

Data Management

Data is stored in a variety of formats and within a variety of database systems, and needs to be accessed in a timely manner and potentially multiple times. Smart caching and other optimisations, tuned for particular classes of analysis algorithms, is required. The semantic extension framework (SEF) for Java (Marquez, Zigman and Blackburn 1999) is an abstraction tool which provides orthogonality of the algorithms with respect to the data source. This approach allows datasets to be transparently accessed and efficiently managed from any source. Algorithms accessing the data simply view the data as Java data structures which are efficiently instantiated as required and as determined by the semantic extensions provide for the relevant objects. This new project is exploring the use of the SEF to provide orthogonality and optimised access to large scale datasets.

Pulling it Together

The Java-based Data Miner's Arcade (Williams 1998) is a platform-independent system for integrating multiple analysis and visualisation tools with a consistent user interface, providing seamless access to data stored in a multitude of systems. Standard interfaces are used where available and data is accessed through ODBC and JDBC or from other sources and managed internally within the Arcade. This is proposed to be redeveloped using the Java semantic extension framework to provide orthogonality between the analysis algorithms and the data sources. The Extensible Markup Language (XML) is adopted as the target "language" for the data mining tools within the environment (Grossman, Bailey, Ramu, Malhi, Hallstrom, Pulleyn and Qin 1999). Models expressed in XML can be visualised, run, or combined with other models in ensemble systems, through the use of plug-ins within the Data Miner's Arcade.

Acknowledgments

References

Bakin, S., Hegland, M., Howard, P. and Williams, G.: 1999, Mining taxation data with parallel BMARS, *Submitted for publication* .

Friedman, J.: 1991, Multivariate adaptive regression splines, *The Annals of Statistics* **19**(1), 1–141.

Grossman, R., Bailey, S., Ramu, A., Malhi, B., Hallstrom, P., Pulleyn, I. and Qin, X.: 1999, The management and mining of multiple predictive models using the predictive modelling markup language, *Information and Software Technology* **41**(9).

Marquez, A., Zigman, J. and Blackburn, S.: 1999, Fast, portable orthogonally persistent java using semi dynamic semantic extensions, *Submitted for publication* .

Nagappan, R.: 1999, Visualising multidimensional non-geometric data sets, *Submitted for publication* .

Williams, G.: 1998, The data miner's arcade, http://www.cmis.csiro.au/Graham.Williams/dataminer/Arcade.html.

Williams, G. J.: 1999, Evolutionary hot spots data mining, *Advances in Data Mining (PAKDD99)*, Lecture Notes in Computer Science, Springer-Verlag.

Williams, G. J. and Huang, Z.: 1997, Mining the knowledge mine: The Hot Spots methodology for mining large, real world databases, *in* A. Sattar (ed.), *Advanced Topics in Artificial Intelligence (AI97)*, Vol. 1342 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 340–348.

Part 1:
Java™ Media Framework

1

Working with
Time-Based Media

Any data that changes meaningfully with respect to time can be characterized as time-based media. Audio clips, MIDI sequences, movie clips, and animations are common forms of time-based media. Such media data can be obtained from a variety of sources, such as local or network files, cameras, microphones, and live broadcasts.

This chapter describes the key characteristics of time-based media and describes the use of time-based media in terms of a fundamental data processing model:

Input	Process	Output
↑ Capture	▦ Apply effect filters	◀)) Present
▤ Read from a file	A ⟷ A Compress or decompress	▤ Save to a file
(NET) Receive from the network	A → B Convert between formats	(NET) Send across the network
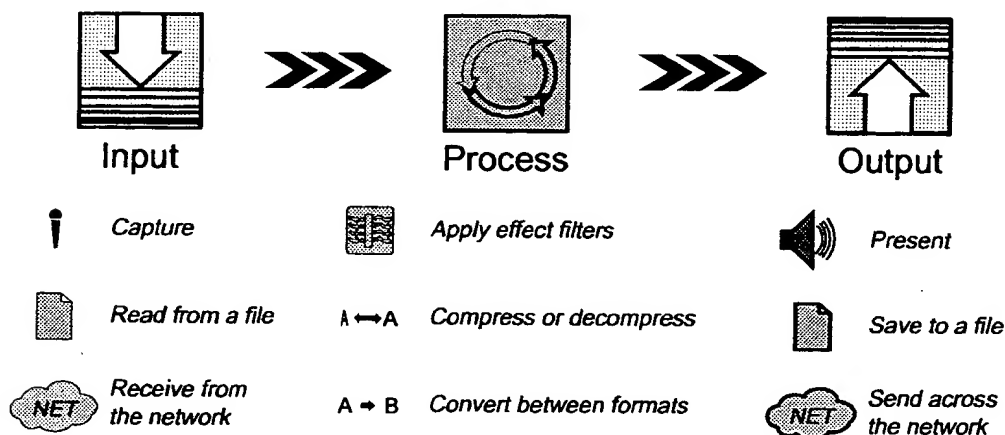
Figure 1-1: Media processing model.

Streaming Media

A key characteristic of time-based media is that it requires timely delivery and processing. Once the flow of media data begins, there are strict timing deadlines that must be met, both in terms of receiving and presenting the data. For this reason, time-based media is often referred to as *streaming media*—it is delivered in a steady stream that must be received and processed within a particular timeframe to produce acceptable results.

For example, when a movie is played, if the media data cannot be delivered quickly enough, there might be odd pauses and delays in playback. On the other hand, if the data cannot be received and processed quickly enough, the movie might appear jumpy as data is lost or frames are intentionally dropped in an attempt to maintain the proper playback rate.

Content Type

The format in which the media data is stored is referred to as its *content type*. QuickTime, MPEG, and WAV are all examples of content types. Content type is essentially synonymous with file type—content type is used because media data is often acquired from sources other than local files.

Media Streams

A *media stream* is the media data obtained from a local file, acquired over the network, or captured from a camera or microphone. Media streams often contain multiple channels of data called *tracks*. For example, a Quicktime file might contain both an audio track and a video track. Media streams that contain multiple tracks are often referred to as *multiplexed* or *complex* media streams. *Demultiplexing* is the process of extracting individual tracks from a complex media stream.

A track's *type* identifies the kind of data it contains, such as audio or video. The *format* of a track defines how the data for the track is structured.

A media stream can be identified by its location and the protocol used to access it. For example, a URL might be used to describe the location of a QuickTime file on a local or remote system. If the file is local, it can be accessed through the FILE protocol. On the other hand, if it's on a web server, the file can be accessed through the HTTP protocol. A *media locator* provides a way to identify the location of a media stream when a URL can't be used.

Media streams can be categorized according to how the data is delivered:

- Pull—data transfer is initiated and controlled from the client side. For example, Hypertext Transfer Protocol (HTTP) and FILE are pull protocols.

- Push—the server initiates data transfer and controls the flow of data. For example, Real-time Transport Protocol (RTP) is a push protocol used for streaming media. Similarly, the SGI MediaBase protocol is a push protocol used for video-on-demand (VOD).

Common Media Formats

The following tables identify some of the characteristics of common media formats. When selecting a format, it's important to take into account the characteristics of the format, the target environment, and the expectations of the intended audience. For example, if you're delivering media content via the web, you need to pay special attention to the bandwidth requirements.

The CPU Requirements column characterizes the processing power necessary for optimal presentation of the specified format. The Bandwidth Requirements column characterizes the transmission speeds necessary to send or receive data quickly enough for optimal presentation.

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
Cinepak	AVI QuickTime	Medium	Low	High
MPEG-1	MPEG	High	High	High
H.261	AVI RTP	Low	Medium	Medium
H.263	QuickTime AVI RTP	Medium	Medium	Low
JPEG	QuickTime AVI RTP	High	High	High

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
Indeo	QuickTime AVI	Medium	Medium	Medium

Table 1-1: Common video formats.

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
PCM	AVI QuickTime WAV	High	Low	High
Mu-Law	AVI QuickTime WAV RTP	Low	Low	High
ADPCM (DVI, IMA4)	AVI QuickTime WAV RTP	Medium	Medium	Medium
MPEG-1	MPEG	High	High	High
MPEG Layer3	MPEG	High	High	Medium
GSM	WAV RTP	Low	Low	Low
G.723.1	WAV RTP	Medium	Medium	Low

Table 1-2: Common audio formats.

Some formats are designed with particular applications and requirements in mind. High-quality, high-bandwidth formats are generally targeted toward CD-ROM or local storage applications. H.261 and H.263 are generally used for video conferencing applications and are optimized for video where there's not a lot of action. Similarly, G.723 is typically used to produce low bit-rate speech for telephony applications.

Media Presentation

Most time-based media is audio or video data that can be presented through output devices such as speakers and monitors. Such devices are the most common *destination* for media data output. Media streams can also be sent to other destinations—for example, saved to a file or transmitted across the network. An output destination for media data is sometimes referred to as a *data sink*.

Presentation Controls

While a media stream is being presented, VCR-style presentation controls are often provided to enable the user to control playback. For example, a control panel for a movie player might offer buttons for stopping, starting, fast-forwarding, and rewinding the movie.

Latency

In many cases, particularly when presenting a media stream that resides on the network, the presentation of the media stream cannot begin immediately. The time it takes before presentation can begin is referred to as the *start latency*. Users might experience this as a delay between the time that they click the start button and the time when playback actually starts.

Multimedia presentations often combine several types of time-based media into a synchronized presentation. For example, background music might be played during an image slide-show, or animated text might be synchronized with an audio or video clip. When the presentation of multiple media streams is synchronized, it is essential to take into account the start latency of each stream—otherwise the playback of the different streams might actually begin at different times.

Presentation Quality

The quality of the presentation of a media stream depends on several factors, including:

- The compression scheme used
- The processing capability of the playback system
- The bandwidth available (for media streams acquired over the network)

Traditionally, the higher the quality, the larger the file size and the greater the processing power and bandwidth required. Bandwidth is usually represented as the number of bits that are transmitted in a certain period of time—the *bit rate.*

To achieve high-quality video presentations, the number of frames displayed in each period of time (the *frame rate*) should be as high as possible. Usually movies at a frame rate of 30 frames-per-second are considered indistinguishable from regular TV broadcasts or video tapes.

Media Processing

In most instances, the data in a media stream is manipulated before it is presented to the user. Generally, a series of processing operations occur before presentation:

1. If the stream is multiplexed, the individual tracks are extracted.

2. If the individual tracks are compressed, they are decoded.

3. If necessary, the tracks are converted to a different format.

4. Effect filters are applied to the decoded tracks (if desired).

The tracks are then delivered to the appropriate output device. If the media stream is to be stored instead of rendered to an output device, the processing stages might differ slightly. For example, if you wanted to capture audio and video from a video camera, process the data, and save it to a file:

1. The audio and video tracks would be captured.

2. Effect filters would be applied to the raw tracks (if desired).

3. The individual tracks would be encoded.

4. The compressed tracks would be multiplexed into a single media stream.

5. The multiplexed media stream would then be saved to a file.

Demultiplexers and Multiplexers

A demultiplexer extracts individual tracks of media data from a multiplexed media stream. A *mutliplexer* performs the opposite function, it takes individual tracks of media data and merges them into a single multiplexed media stream.

Codecs

A codec performs media-data compression and decompression. When a track is encoded, it is converted to a compressed format suitable for storage or transmission; when it is decoded it is converted to a non-compressed (raw) format suitable for presentation.

Each codec has certain input formats that it can handle and certain output formats that it can generate. In some situations, a series of codecs might be used to convert from one format to another.

Effect Filters

An effect filter modifies the track data in some way, often to create special effects such as blur or echo.

Effect filters are classified as either pre-processing effects or post-processing effects, depending on whether they are applied before or after the codec processes the track. Typically, effect filters are applied to uncompressed (raw) data.

Renderers

A renderer is an abstraction of a presentation device. For audio, the presentation device is typically the computer's hardware audio card that outputs sound to the speakers. For video, the presentation device is typically the computer monitor.

Compositing

Certain specialized devices support *compositing*. Compositing time-based media is the process of combining multiple tracks of data onto a single presentation medium. For example, overlaying text on a video presentation is one common form of compositing. Compositing can be done in either hardware or software. A device that performs compositing can be abstracted as a renderer that can receive multiple tracks of input data.

Media Capture

Time-based media can be captured from a live source for processing and playback. For example, audio can be captured from a microphone or a video capture card can be used to obtain video from a camera. Capturing can be thought of as the *input* phase of the standard media processing model.

A capture device might deliver multiple media streams. For example, a video camera might deliver both audio and video. These streams might be captured and manipulated separately or combined into a single, multiplexed stream that contains both an audio track and a video track.

Capture Devices

To capture time-based media you need specialized hardware—for example, to capture audio from a live source, you need a microphone and an appropriate audio card. Similarly, capturing a TV broadcast requires a TV tuner and an appropriate video capture card. Most systems provide a query mechanism to find out what capture devices are available.

Capture devices can be characterized as either push or pull sources. For example, a still camera is a pull source—the user controls when to capture an image. A microphone is a push source—the live source continuously provides a stream of audio.

The format of a captured media stream depends on the processing performed by the capture device. Some devices do very little processing and deliver raw, uncompressed data. Other capture devices might deliver the data in a compressed format.

Capture Controls

Controls are sometimes provided to enable the user to manage the capture process. For example, a capture control panel might enable the user to specify the data rate and encoding type for the captured stream and start and stop the capture process.

2
Understanding JMF

Java™ Media Framework (JMF) provides a unified architecture and messaging protocol for managing the acquisition, processing, and delivery of time-based media data. JMF is designed to support most standard media content types, such as AIFF, AU, AVI, GSM, MIDI, MPEG, QuickTime, RMF, and WAV.

By exploiting the advantages of the Java platform, JMF delivers the promise of "Write Once, Run Anywhere™" to developers who want to use media such as audio and video in their Java programs. JMF provides a common cross-platform Java API for accessing underlying media frameworks. JMF implementations can leverage the capabilities of the underlying operating system, while developers can easily create portable Java programs that feature time-based media by writing to the JMF API.

With JMF, you can easily create applets and applications that present, capture, manipulate, and store time-based media. The framework enables advanced developers and technology providers to perform custom processing of raw media data and seamlessly extend JMF to support additional content types and formats, optimize handling of supported formats, and create new presentation mechanisms.

High-Level Architecture

Devices such as tape decks and VCRs provide a familiar model for recording, processing, and presenting time-based media. When you play a movie using a VCR, you provide the media stream to the VCR by inserting a video tape. The VCR reads and interprets the data on the tape and sends appropriate signals to your television and speakers.
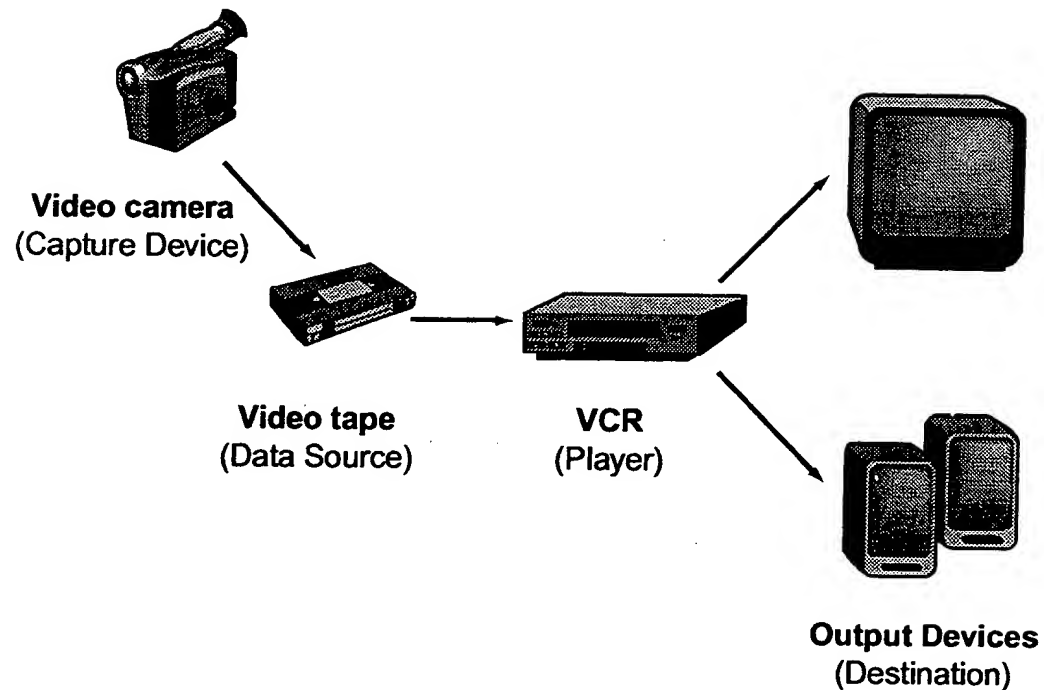
Figure 2-1: Recording, processing, and presenting time-based media.

JMF uses this same basic model. A *data source* encapsulates the media stream much like a video tape and a *player* provides processing and control mechanisms similar to a VCR. Playing and capturing audio and video with JMF requires the appropriate input and output devices such as microphones, cameras, speakers, and monitors.

Data sources and players are integral parts of JMF's high-level API for managing the capture, presentation, and processing of time-based media. JMF also provides a lower-level API that supports the seamless integration of custom processing components and extensions. This layering provides Java developers with an easy-to-use API for incorporating time-based media into Java programs while maintaining the flexibility and extensibility required to support advanced media applications and future media technologies.
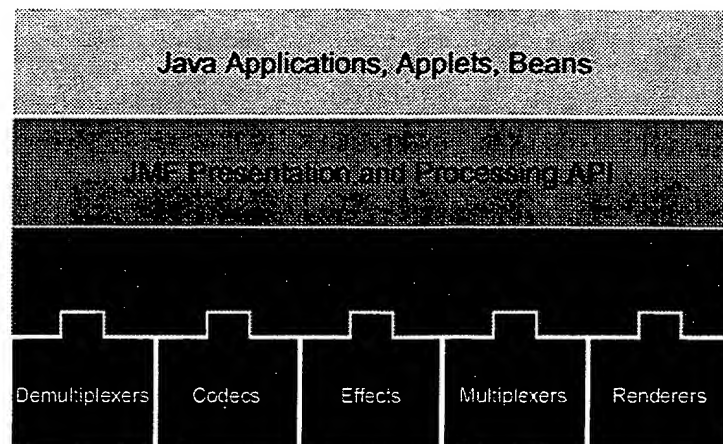
Figure 2-2: High-level JMF achitecture.

Time Model

JMF keeps time to nanosecond precision. A particular point in time is typically represented by a Time object, though some classes also support the specification of time in nanoseconds.

Classes that support the JMF time model implement Clock to keep track of time for a particular media stream. The Clock interface defines the basic timing and synchronization operations that are needed to control the presentation of media data.
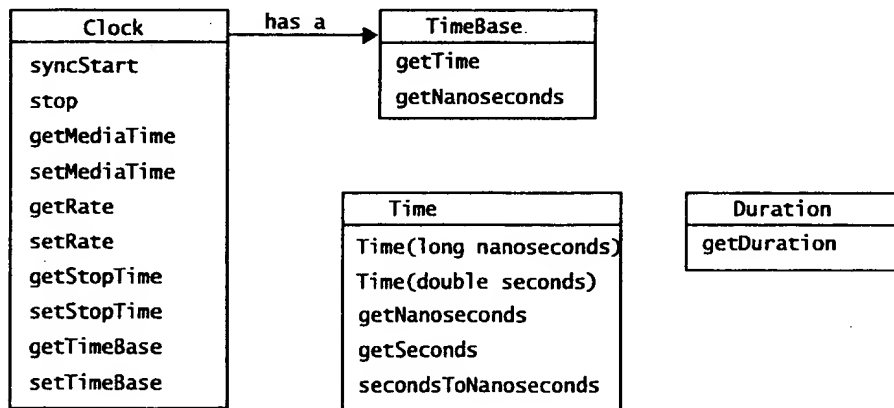
Figure 2-3: JMF time model.

A Clock uses a TimeBase to keep track of the passage of time while a media stream is being presented. A TimeBase provides a constantly ticking time source, much like a crystal oscillator in a watch. The only information that a TimeBase provides is its current time, which is referred to as the *time-base*

time. The time-base time cannot be stopped or reset. Time-base time is often based on the system clock.

A Clock object's *media time* represents the current position within a media stream—the beginning of the stream is media time zero, the end of the stream is the maximum media time for the stream. The *duration* of the media stream is the elapsed time from start to finish—the length of time that it takes to present the media stream. (Media objects implement the Duration interface if they can report a media stream's duration.)

To keep track of the current media time, a Clock uses:

- The time-base start-time—the time that its TimeBase reports when the presentation begins.

- The media start-time—the position in the media stream where presentation begins.

- The playback rate—how fast the Clock is running in relation to its TimeBase. The *rate* is a scale factor that is applied to the TimeBase. For example, a rate of 1.0 represents the normal playback rate for the media stream, while a rate of 2.0 indicates that the presentation will run at twice the normal rate. A negative rate indicates that the Clock is running in the opposite direction from its TimeBase—for example, a negative rate might be used to play a media stream backward.

When presentation begins, the media time is mapped to the time-base time and the advancement of the time-base time is used to measure the passage of time. During presentation, the current media time is calculated using the following formula:

```
MediaTime = MediaStartTime + Rate(TimeBaseTime - TimeBaseStartTime)
```

When the presentation stops, the media time stops, but the time-base time continues to advance. If the presentation is restarted, the media time is remapped to the current time-base time.

Managers

The JMF API consists mainly of interfaces that define the behavior and interaction of objects used to capture, process, and present time-based media. Implementations of these interfaces operate within the structure of the framework. By using intermediary objects called *managers*, JMF makes it easy to integrate new implementations of key interfaces that can be used seamlessly with existing classes.

JMF uses four managers:

- `Manager`—handles the construction of `Players`, `Processors`, `DataSources`, and `DataSinks`. This level of indirection allows new implementations to be integrated seamlessly with JMF. From the client perspective, these objects are always created the same way whether the requested object is constructed from a default implementation or a custom one.

- `PackageManager`—maintains a registry of packages that contain JMF classes, such as custom `Players`, `Processors`, `DataSources`, and `DataSinks`.

- `CaptureDeviceManager`—maintains a registry of available capture devices.

- `PlugInManager`—maintains a registry of available JMF plug-in processing components, such as `Multiplexers`, `Demultiplexers`, `Codecs`, `Effects`, and `Renderers`.

To write programs based on JMF, you'll need to use the `Manager` create methods to construct the `Players`, `Processors`, `DataSources`, and `DataSinks` for your application. If you're capturing media data from an input device, you'll use the `CaptureDeviceManager` to find out what devices are available and access information about them. If you're interested in controlling what processing is performed on the data, you might also query the `PlugInManager` to determine what plug-ins have been registered.

If you extend JMF functionality by implementing a new plug-in, you can register it with the `PlugInManager` to make it available to `Processors` that support the plug-in API. To use a custom `Player`, `Processor`, `DataSource`, or `DataSink` with JMF, you register your unique package prefix with the `PackageManager`.

Event Model

JMF uses a structured event reporting mechanism to keep JMF-based programs informed of the current state of the media system and enable JMF-based programs to respond to media-driven error conditions, such as out-of data and resource unavailable conditions. Whenever a JMF object needs to report on the current conditions, it posts a `MediaEvent`. `MediaEvent` is subclassed to identify many particular types of events. These objects follow the established Java Beans patterns for events.

For each type of JMF object that can post MediaEvents, JMF defines a corresponding listener interface. To receive notification when a MediaEvent is posted, you implement the appropriate listener interface and register your listener class with the object that posts the event by calling its addListener method.

Controller objects (such as Players and Processors) and certain Control objects such as GainControl post media events.
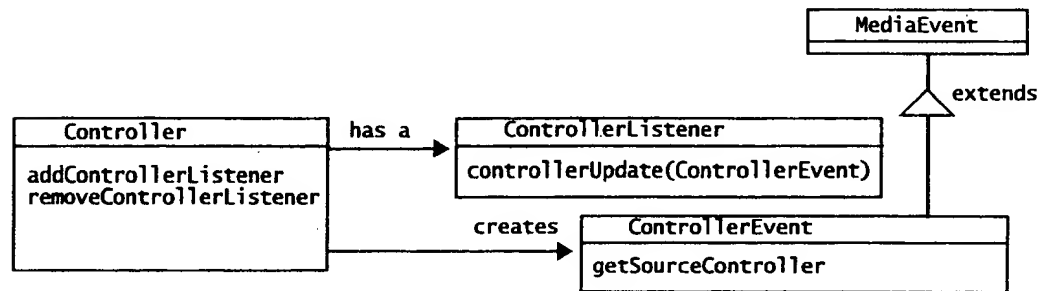
Figure 2-4: JMF event model.

RTPSessionManager objects also post events. For more information, see "RTP Events" on page 122.

Data Model

JMF media players usually use DataSources to manage the transfer of media-content. A DataSource encapsulates both the location of media and the protocol and software used to deliver the media. Once obtained, the source cannot be reused to deliver other media.

A DataSource is identified by either a JMF MediaLocator or a URL (universal resource locator). A MediaLocator is similar to a URL and can be constructed from a URL, but can be constructed even if the corresponding protocol handler is not installed on the system. (In Java, a URL can only be constructed if the corresponding protocol handler is installed on the system.)

A DataSource manages a set of SourceStream objects. A standard data source uses a byte array as the unit of transfer. A *buffer data source* uses a Buffer object as its unit of transfer. JMF defines several types of DataSource objects:

be repositioned and a client program could allow the user to replay the video clip or seek to a new position in the video. In contrast, broadcast media is under server control and cannot be repositioned. Some VOD protocols might support limited user control—for example, a client program might be able to allow the user to seek to a new position, but not fast forward or rewind.

Specialty DataSources

JMF defines two types of specialty data sources, cloneable data sources and merging data sources.

A cloneable data source can be used to create clones of either a pull or push DataSource. To create a cloneable DataSource, you call the Manager createCloneableDataSource method and pass in the DataSource you want to clone. Once a DataSource has been passed to createCloneableData-Source, you should only interact with the cloneable DataSource and its clones; the original DataSource should no longer be used directly.

Cloneable data sources implement the SourceCloneable interface, which defines one method, createClone. By calling createClone, you can create any number of clones of the DataSource that was used to construct the cloneable DataSource. The clones can be controlled through the cloneable DataSource used to create them— when connect, disconnect, start, or stop is called on the cloneable DataSource, the method calls are propagated to the clones.

The clones don't necessarily have the same properties as the cloneable data source used to create them or the original DataSource. For example, a cloneable data source created for a capture device might function as a master data source for its clones—in this case, unless the cloneable data source is used, the clones won't produce any data. If you hook up both the cloneable data source and one or more clones, the clones will produce data at the same rate as the master.

A MergingDataSource can be used to combine the SourceStreams from several DataSources into a single DataSource. This enables a set of Data-Sources to be managed from a single point of control—when connect, disconnect, start, or stop is called on the MergingDataSource, the method calls are propagated to the merged DataSources.

To construct a MergingDataSource, you call the Manager createMerging-DataSource method and pass in an array that contains the data sources you want to merge. To be merged, all of the DataSources must be of the

same type; for example, you cannot merge a PullDataSource and a Push-DataSource. The duration of the merged DataSource is the maximum of the merged DataSource objects' durations. The ContentType is application/mixed-media.

Data Formats

The exact media format of an object is represented by a Format object. The format itself carries no encoding-specific parameters or global timing information, it describes the format's encoding name and the type of data the format requires.

JMF extends Format to define audio- and video-specific formats.
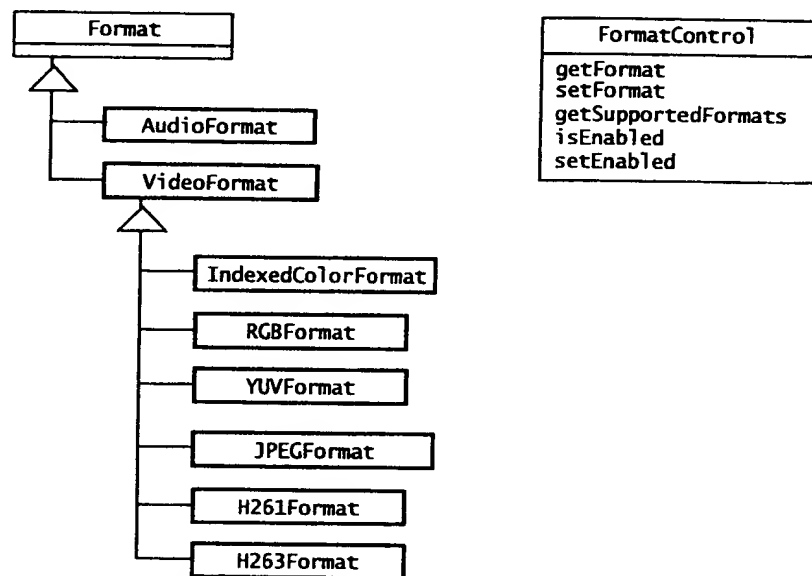
Figure 2-6: JMF media formats.

An AudioFormat describes the attributes specific to an audio format, such as sample rate, bits per sample, and number of channels. A VideoFormat encapsulates information relevant to video data. Several formats are derived from VideoFormat to describe the attributes of common video formats, including:

- IndexedColorFormat
- RGBFormat
- YUVFormat
- JPEGFormat
- H261Format
- H263Format

To receive notification of format changes from a `Controller`, you implement the `ControllerListener` interface and listen for `FormatChangeEvents`. (For more information, see "Responding to Media Events" on page 54.)

Controls

JMF `Control` provides a mechanism for setting and querying attributes of an object. A `Control` often provides access to a corresponding user interface component that enables user control over an object's attributes. Many JMF objects expose `Controls`, including `Controller` objects, `DataSource` objects, `DataSink` objects, and JMF plug-ins.

Any JMF object that wants to provide access to its corresponding `Control` objects can implement the `Controls` interface. `Controls` defines methods for retrieving associated `Control` objects. `DataSource` and `PlugIn` use the `Controls` interface to provide access to their `Control` objects.

Standard Controls

JMF defines the standard `Control` interfaces shown in Figure 2-8:, "JMF controls."

`CachingControl` enables download progress to be monitored and displayed. If a `Player` or `Processor` can report its download progress, it implements this interface so that a progress bar can be displayed to the user.

`GainControl` enables audio volume adjustments such as setting the level and muting the output of a `Player` or `Processor`. It also supports a listener mechanism for volume changes.
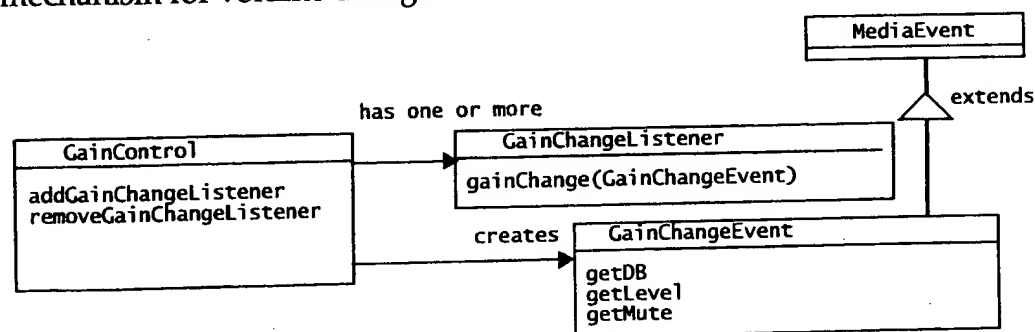
Figure 2-7: Gain control.
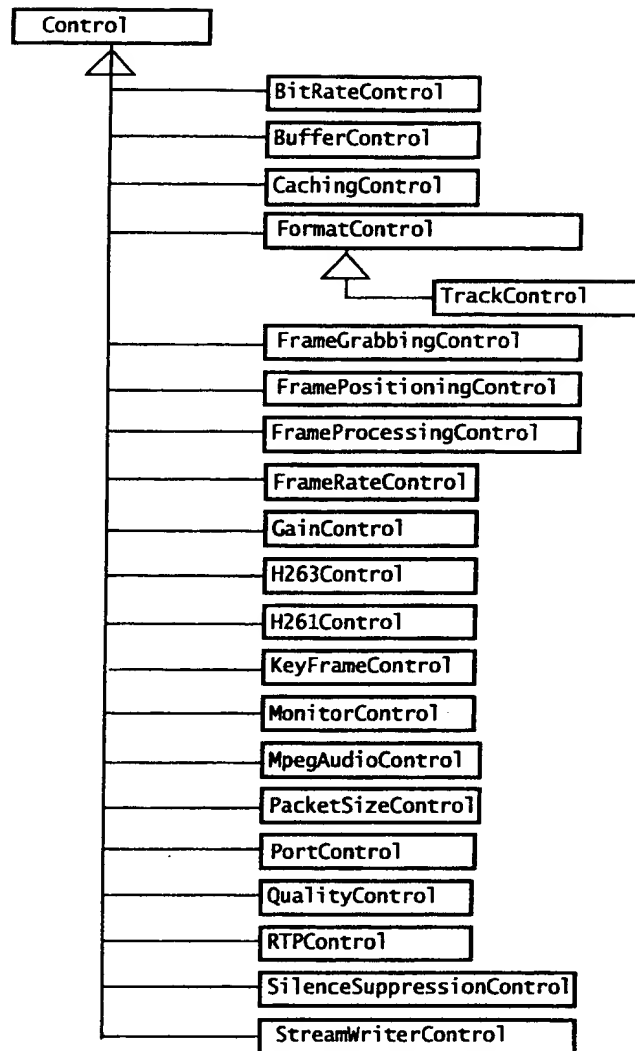
Figure 2-8: JMF controls.

DataSink or Multiplexer objects that read media from a DataSource and write it out to a destination such as a file can implement the StreamWriterControl interface. This Control enables the user to limit the size of the stream that is created.

FramePositioningControl and FrameGrabbingControl export frame-based capabilities for Players and Processors. FramePositioningControl enables precise frame positioning within a Player or Processor object's media stream. FrameGrabbingControl provides a mechanism for grabbing a still video frame from the video stream. The FrameGrabbingControl can also be supported at the Renderer level.

Objects that have a Format can implement the FormatControl interface to provide access to the Format. FormatControl also provides methods for querying and setting the format.

A TrackControl is a type of FormatControl that provides the mechanism for controlling what processing a Processor object performs on a particular track of media data. With the TrackControl methods, you can specify what format conversions are performed on individual tracks and select the Effect, Codec, or Renderer plug-ins that are used by the Processor. (For more information about processing media data, see "Processing Time-Based Media with JMF" on page 71.)

Two controls, PortControl and MonitorControl enable user control over the capture process. PortControl defines methods for controlling the output of a capture device. MonitorControl enables media data to be previewed as it is captured or encoded.

BufferControl enables user-level control over the buffering done by a particular object.

JMF also defines several codec controls to enable control over hardware or software encoders and decoders:

- BitRateControl—used to export the bit rate information for an incoming stream or to control the encoding bit rate. Enables specification of the bit rate in bits per second.

- FrameProcessingControl—enables the specification of frame processing parameters that allow the codec to perform minimal processing when it is falling behind on processing the incoming data.

- FrameRateControl—enables modification of the frame rate.

- H261Control—enables control over the H.261 video codec still-image transmission mode.

- H263Control—enables control over the H.263 video-codec parameters, including support for the unrestricted vector, arithmetic coding, advanced prediction, PB Frames, and error compensation extensions.

- KeyFrameControl—enables the specification of the desired interval between key frames. (The encoder can override the specified key-frame interval if necessary.)

- MpegAudioControl—exports an MPEG audio codec's capabilities and enables the specification of selected MPEG encoding parameters.

- QualityControl—enables specification of a preference in the trade-off

between quality and CPU usage in the processing performed by a codec. This quality hint can have different effects depending on the type of compression. A higher quality setting will result in better quality of the resulting bits, for example better image quality for video.

- SilenceSuppressionControl—enables specification of silence suppression parameters for audio codecs. When silence suppression mode is on, an audio encoder does not output any data if it detects silence at its input.

User Interface Components

A Control can provide access to a user interface Component that exposes its control behavior to the end user. To get the default user interface component for a particular Control, you call getControlComponent. This method returns an AWT Component that you can add to your applet's presentation space or application window.

A Controller might also provide access to user interface Components. For example, a Player provides access to both a visual component and a control panel component—to retrieve these components, you call the Player methods getVisualComponent and getControlPanelComponent.

If you don't want to use the default control components provided by a particular implementation, you can implement your own and use the event listener mechanism to determine when they need to be updated. For example, you might implement your own GUI components that support user interaction with a Player. Actions on your GUI components would trigger calls to the appropriate Player methods, such as start and stop. By registering your custom GUI components as ControllerListeners for the Player, you can also update your GUI in response to changes in the Player object's state.

Extensibility

Advanced developers and technology providers can extend JMF functionality in two ways:

- By implementing custom processing components (*plug-ins*) that can be interchanged with the standard processing components used by a JMF Processor

- By directly implementing the Controller, Player, Processor, DataSource, or DataSink interfaces

Implementing a JMF plug-in enables you to customize or extend the capabilities of a Processor without having to implement one from scratch. Once a plug-in is registered with JMF, it can be selected as a processing option for any Processor that supports the plug-in API. JMF plug-ins can be used to:

- Extend or replace a Processor object's processing capability piecewise by selecting the individual plug-ins to be used.

- Access the media data at specific points in the data flow. For example, different Effect plug-ins can be used for pre- and post-processing of the media data associated with a Processor.

- Process media data outside of a Player or Processor. For example, you might use a Demultiplexer plug-in to get individual audio tracks from a multiplexed media-stream so you could play the tracks through Java Sound.

In situations where an even greater degree of flexibility and control is required, custom implementations of the JMF Controller, Player, Processor, DataSource, or DataSink interfaces can be developed and used seamlessly with existing implementations. For example, if you have a hardware MPEG decoder, you might want to implement a Player that takes input from a DataSource and uses the decoder to perform the parsing, decoding, and rendering all in one step. Custom Players and Processors can also be implemented to integrate media engines such as Microsoft's Media Player, Real Network's RealPlayer, and IBM's HotMedia with JMF.

Note: JMF Players and Processors are not required to support plug-ins. Plug-ins won't work with JMF 1.0-based Players and some Processor implementations might choose not to support them. The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation fully supports the plug-in API.

Presentation

In JMF, the presentation process is modeled by the Controller interface. Controller defines the basic state and control mechanism for an object that controls, presents, or captures time-based media. It defines the phases

A Player does not provide any control over the processing that it performs or how it renders the media data.

Player supports standardized user control and relaxes some of the operational restrictions imposed by Clock and Controller.
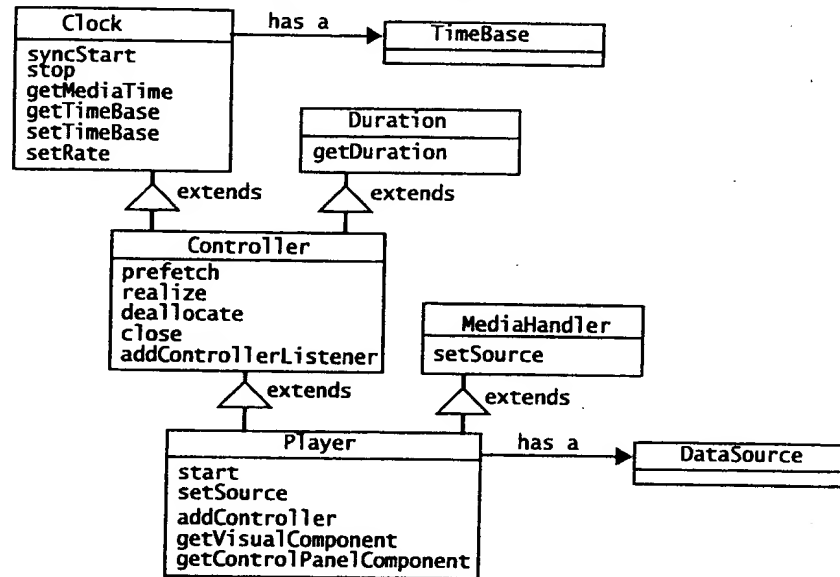
Figure 2-11: JMF players.

Player States

A Player can be in one of six states. The Clock interface defines the two primary states: *Stopped* and *Started*. To facilitate resource management, Controller breaks the *Stopped* state down into five standby states: *Unrealized, Realizing, Realized, Prefetching,* and *Prefetched.*
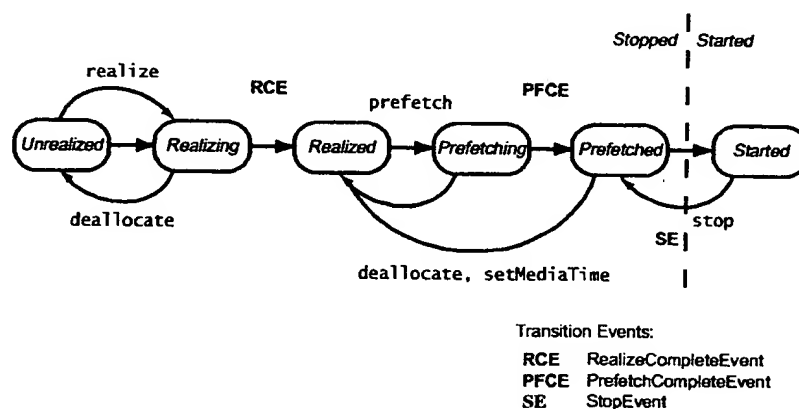
Figure 2-12: Player states.

In normal operation, a Player steps through each state until it reaches the *Started* state:

- A Player in the *Unrealized* state has been instantiated, but does not yet know anything about its media. When a media Player is first created, it is *Unrealized.*

- When realize is called, a Player moves from the *Unrealized* state into the *Realizing* state. A *Realizing* Player is in the process of determining its resource requirements. During realization, a Player acquires the resources that it only needs to acquire once. These might include rendering resources other than exclusive-use resources. (Exclusive-use resources are limited resources such as particular hardware devices that can only be used by one Player at a time; such resources are acquired during *Prefetching.*) A *Realizing* Player often downloads assets over the network.

- When a Player finishes *Realizing,* it moves into the *Realized* state. A *Realized* Player knows what resources it needs and information about the type of media it is to present. Because a *Realized* Player knows how to render its data, it can provide visual components and controls. Its connections to other objects in the system are in place, but it does not own any resources that would prevent another Player from starting.

- When prefetch is called, a Player moves from the *Realized* state into the *Prefetching* state. A *Prefetching* Player is preparing to present its media. During this phase, the Player preloads its media data, obtains exclusive-use resources, and does whatever else it needs to do to prepare itself to play. *Prefetching* might have to recur if a Player object's media presentation is repositioned, or if a change in the Player object's rate requires that additional buffers be acquired or alternate processing take place.

- When a Player finishes *Prefetching,* it moves into the *Prefetched* state. A *Prefetched* Player is ready to be started.

- Calling start puts a Player into the *Started* state. A *Started* Player object's time-base time and media time are mapped and its clock is running, though the Player might be waiting for a particular time to begin presenting its media data.

A Player posts TransitionEvents as it moves from one state to another. The ControllerListener interface provides a way for your program to determine what state a Player is in and to respond appropriately. For example, when your program calls an asynchronous method on a Player

or `Processor`, it needs to listen for the appropriate event to determine when the operation is complete.

Using this event reporting mechanism, you can manage a `Player` object's start latency by controlling when it begins *Realizing* and *Prefetching*. It also enables you to determine whether or not the `Player` is in an appropriate state before calling methods on the `Player`.

Methods Available in Each Player State

To prevent race conditions, not all methods can be called on a `Player` in every state. The following table identifies the restrictions imposed by JMF. If you call a method that is illegal in a `Player` object's current state, the `Player` throws an error or exception.

Method	Unrealized Player	Realized Player	Prefetched Player	Started Player
addController	NotRealizedError	legal	legal	ClockStartedError
deallocate	legal	legal	legal	ClockStartedError
getControlPanelComponent	NotRealizedError	legal	legal	legal
getGainControl	NotRealizedError	legal	legal	legal
getStartLatency	NotRealizedError	legal	legal	legal
getTimeBase	NotRealizedError	legal	legal	legal
getVisualComponent	NotRealizedError	legal	legal	legal
mapToTimeBase	ClockStoppedException	ClockStoppedException	ClockStoppedException	legal
removeController	NotRealizedError	legal	legal	ClockStartedError
setMediaTime	NotRealizedError	legal	legal	legal
setRate	NotRealizedError	legal	legal	legal
setStopTime	NotRealizedError	legal	legal	StopTimeSetError if previously set
setTimeBase	NotRealizedError	legal	legal	ClockStartedError
syncStart	NotPrefetchedError	NotPrefetchedError	legal	ClockStartedError

Table 2-1: Method restrictions for players.

Processors

Processors can also be used to present media data. A Processor is just a specialized type of Player that provides control over what processing is performed on the input media stream. A Processor supports all of the same presentation controls as a Player.
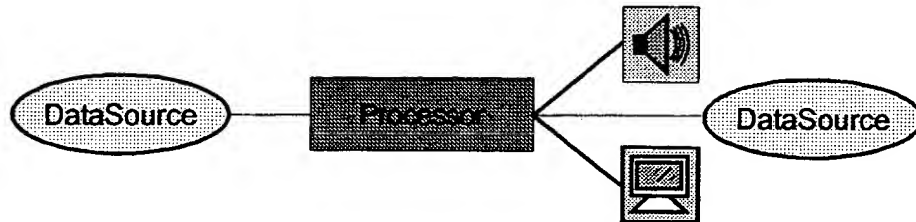
Figure 2-13: JMF processor model.

In addition to rendering media data to presentation devices, a Processor can output media data through a DataSource so that it can be presented by another Player or Processor, further manipulated by another Processor, or delivered to some other destination, such as a file.

For more information about Processors, see "Processing" on page 32.

Presentation Controls

In addition to the standard presentation controls defined by Controller, a Player or Processor might also provide a way to adjust the playback volume. If so, you can retrieve its GainControl by calling getGainControl. A GainControl object posts a GainChangeEvent whenever the gain is modified. By implementing the GainChangeListener interface, you can respond to gain changes. For example, you might want to update a custom gain control Component.

Additional custom Control types might be supported by a particular Player or Processor implementation to provide other control behaviors and expose custom user interface components. You access these controls through the getControls method.

For example, the CachingControl interface extends Control to provide a mechanism for displaying a download progress bar. If a Player can report its download progress, it implements this interface. To find out if a Player supports CachingControl, you can call getControl(CachingControl) or use getControls to get a list of all the supported Controls.

Standard User Interface Components

A Player or Processor generally provides two standard user interface components, a visual component and a control-panel component. You can access these Components directly through the getVisualComponent and get-ControlPanelComponent methods.

You can also implement custom user interface components, and use the event listener mechanism to determine when they need to be updated.

Controller Events

The ControllerEvents posted by a Controller such as a Player or Processor fall into three categories: change notifications, closed events, and transition events:

- Change notification events such as RateChangeEvent, DurationUpdateEvent, and FormatChangeEvent indicate that some attribute of the Controller has changed, often in response to a method call. For example, a Player posts a RateChangeEvent when its rate is changed by a call to setRate.

- TransitionEvents allow your program to respond to changes in a Controller object's state. A Player posts transition events whenever it moves from one state to another. (See "Player States" on page 26 for more information about the states and transitions.)

- ControllerClosedEvents are posted by a Controller when it shuts down. When a Controller posts a ControllerClosedEvent, it is no longer usable. A ControllerErrorEvent is a special case of ControllerClosedEvent. You can listen for ControllerErrorEvents so that your program can respond to Controller malfunctions to minimize the impact on the user.
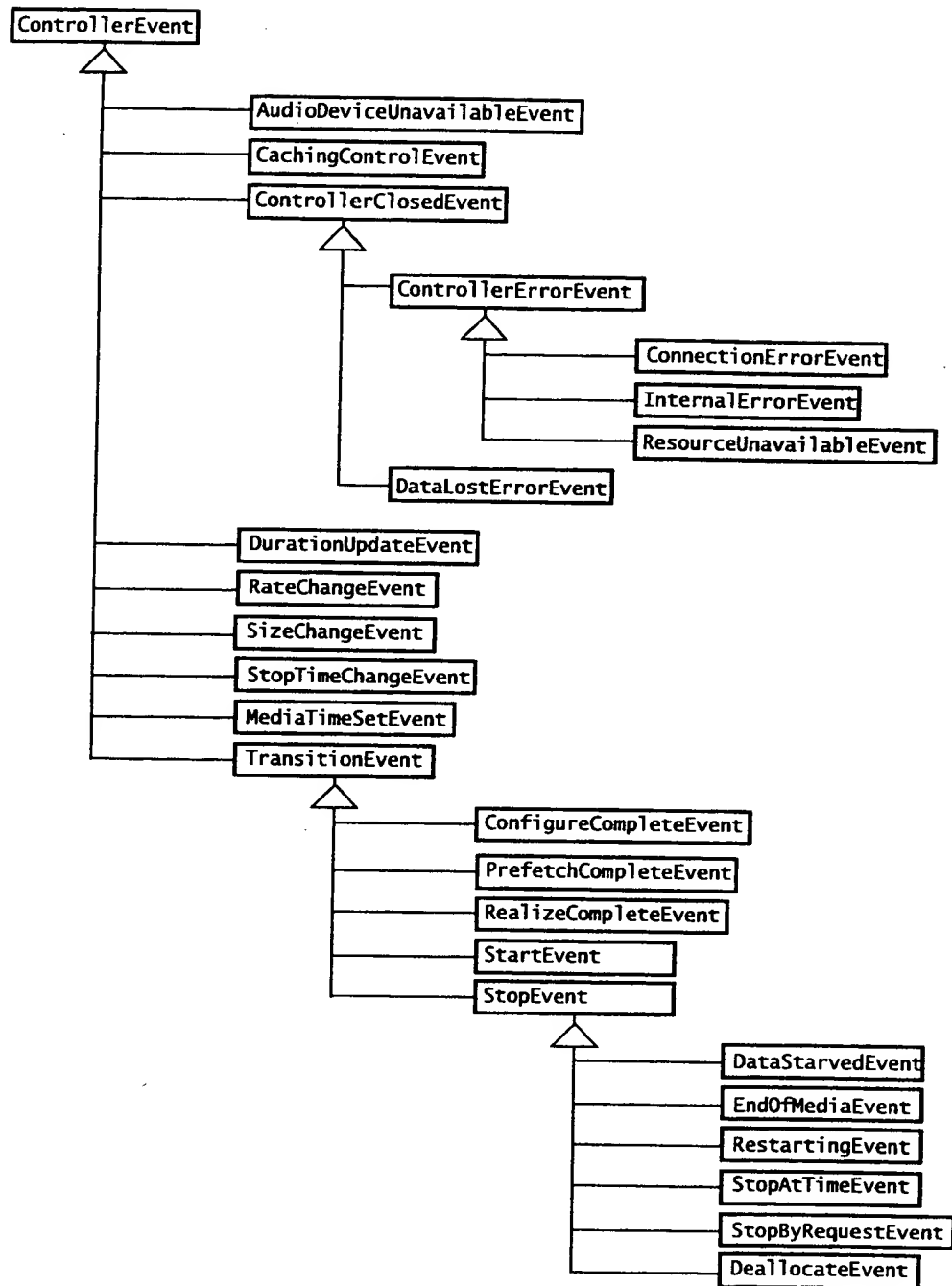
Figure 2-14: JMF events.

Processing

A `Processor` is a `Player` that takes a `DataSource` as input, performs some user-defined processing on the media data, and then outputs the processed media data.
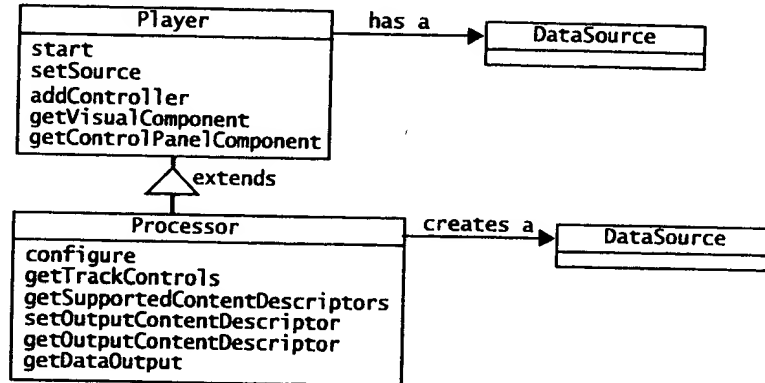
Figure 2-15: JMF processors.

A `Processor` can send the output data to a presentation device or to a `DataSource`. If the data is sent to a `DataSource`, that `DataSource` can be used as the input to another `Player` or `Processor`, or as the input to a `DataSink`.

While the processing performed by a `Player` is predefined by the implementor, a `Processor` allows the application developer to define the type of processing that is applied to the media data. This enables the application of effects, mixing, and compositing in real-time.

The processing of the media data is split into several stages:
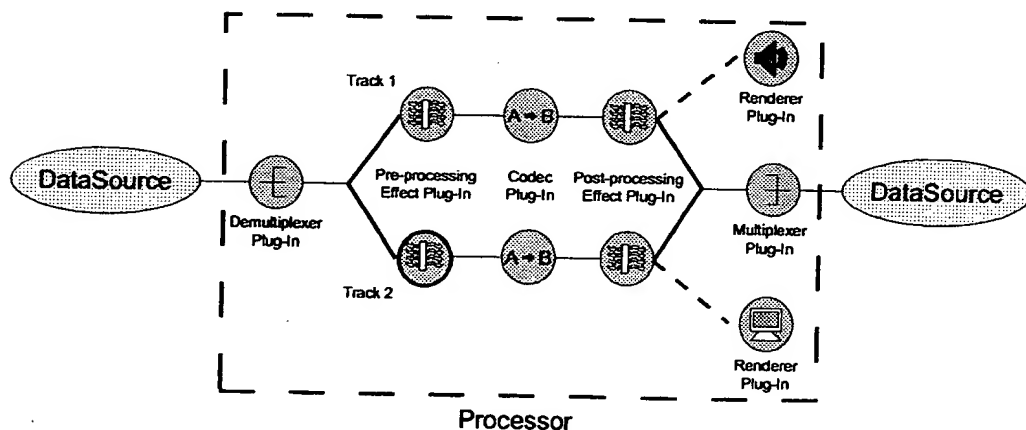
Figure 2-16: Processor stages.

- Demultiplexing is the process of parsing the input stream. If the stream contains multiple tracks, they are extracted and output

separately. For example, a QuickTime file might be demultiplexed into separate audio and video tracks. Demultiplexing is performed automatically whenever the input stream contains multiplexed data.

- Pre-Processing is the process of applying effect algorithms to the tracks extracted from the input stream.

- Transcoding is the process of converting each track of media data from one input format to another. When a data stream is converted from a compressed type to an uncompressed type, it is generally referred to as decoding. Conversely, converting from an uncompressed type to a compressed type is referred to as encoding.

- Post-Processing is the process of applying effect algorithms to decoded tracks.

- Multiplexing is the process of interleaving the transcoded media tracks into a single output stream. For example, separate audio and video tracks might be multiplexed into a single MPEG-1 data stream. You can specify the data type of the output stream with the Processor setOutputContentDescriptor method.

- Rendering is the process of presenting the media to the user.

The processing at each stage is performed by a separate processing component. These processing components are JMF *plug-ins*. If the Processor supports TrackControls, you can select which plug-ins you want to use to process a particular track. There are five types of JMF plug-ins:

- Demultiplexer—parses media streams such as WAV, MPEG or QuickTime. If the stream is multiplexed, the separate tracks are extracted.

- Effect—performs special effects processing on a track of media data.

- Codec—performs data encoding and decoding.

- Multiplexer—combines multiple tracks of input data into a single interleaved output stream and delivers the resulting stream as an output DataSource.

- Renderer—processes the media data in a track and delivers it to a destination such as a screen or speaker.

Processor States

A Processor has two additional standby states, *Configuring* and *Configured*, which occur before the Processor enters the *Realizing* state..
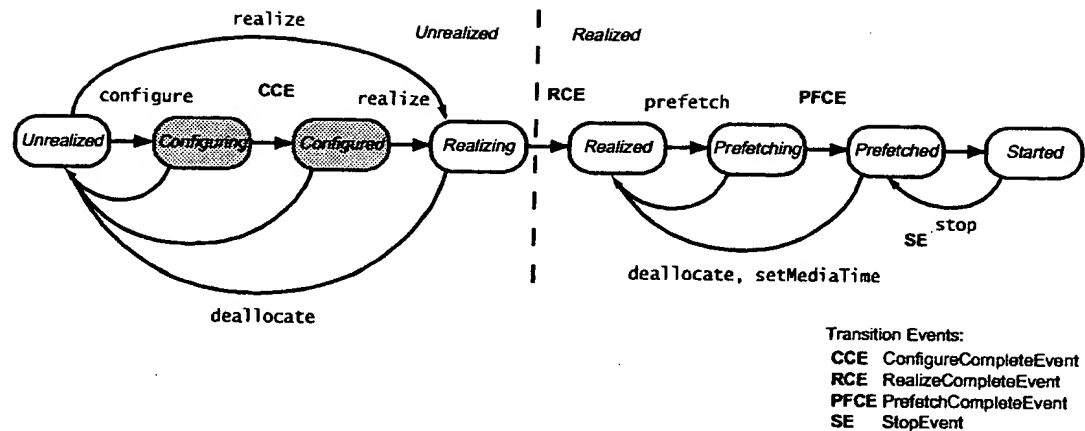
Figure 2-17: Processor states.

- A Processor enters the *Configuring* state when configure is called. While the Processor is in the *Configuring* state, it connects to the DataSource, demultiplexes the input stream, and accesses information about the format of the input data.

- The Processor moves into the *Configured* state when it is connected to the DataSource and data format has been determined. When the Processor reaches the *Configured* state, a ConfigureCompleteEvent is posted.

- When Realize is called, the Processor is transitioned to the Realized state. Once the Processor is *Realized* it is fully constructed.

While a Processor is in the *Configured* state, getTrackControls can be called to get the TrackControl objects for the individual tracks in the media stream. These TrackControl objects enable you specify the media processing operations that you want the Processor to perform.

Calling realize directly on an *Unrealized* Processor automatically transitions it through the *Configuring* and *Configured* states to the *Realized* state. When you do this, you cannot configure the processing options through the TrackControls—the default Processor settings are used.

Calls to the TrackControl methods once the Processor is in the *Realized* state will typically fail, though some Processor implementations might support them.

Methods Available in Each Processor State

Since a `Processor` is a type of `Player`, the restrictions on when methods can be called on a `Player` also apply to `Processors`. Some of the `Processor`-specific methods also are restricted to particular states. The following table shows the restrictions that apply to a `Processor`. If you call a method that is illegal in the current state, the `Processor` throws an error or exception.

Method	Unrealized Processor	Configuring Processor	Configured Processor	Realized Processor
addController	NotRealizedError	NotRealizedError	NotRealizedError	legal
deallocate	legal	legal	legal	legal
getControlPanelComponent	NotRealizedError	NotRealizedError	NotRealizedError	legal
getControls	legal	legal	legal	legal
getDataOutput	NotRealizedError	NotRealizedError	NotRealizedError	legal
getGainControl	NotRealizedError	NotRealizedError	NotRealizedError	legal
getOutputContentDescriptor	NotConfiguredError	NotConfiguredError	legal	legal
getStartLatency	NotRealizedError	NotRealizedError	NotRealizedError	legal
getSupportedContent-Descriptors	legal	legal	legal	legal
getTimeBase	NotRealizedError	NotRealizedError	NotRealizedError	legal
getTrackControls	NotConfiguredError	NotConfiguredError	legal	FormatChange-Exception
getVisualComponent	NotRealizedError	NotRealizedError	NotRealizedError	legal
mapToTimeBase	ClockStoppedException	ClockStoppedException	ClockStoppedException	ClockStopped-Exception
realize	legal	legal	legal	legal
removeController	NotRealizedError	NotRealizedError	NotRealizedError	legal
setOutputContentDescriptor	NotConfiguredError	NotConfiguredError	legal	FormatChange-Exception
setMediaTime	NotRealizedError	NotRealizedError	NotRealizedError	legal
setRate	NotRealizedError	NotRealizedError	NotRealizedError	legal
setStopTime	NotRealizedError	NotRealizedError	NotRealizedError	legal
setTimeBase	NotRealizedError	NotRealizedError	NotRealizedError	legal
syncStart	NotPrefetchedError	NotPrefetchedError	NotPrefetchedError	NotPrefetchedError

Table 2-2: Method restrictions for processors.

Processing Controls

You can control what processing operations the Processor performs on a track through the TrackControl for that track. You call Processor getTrackControls to get the TrackControl objects for all of the tracks in the media stream.

Through a TrackControl, you can explicitly select the Effect, Codec, and Renderer plug-ins you want to use for the track. To find out what options are available, you can query the PlugInManager to find out what plug-ins are installed.

To control the transcoding that's performed on a track by a particular Codec, you can get the Controls associated with the track by calling the TrackControl getControls method. This method returns the codec controls available for the track, such as BitRateControl and QualityControl. (For more information about the codec controls defined by JMF, see "Controls" on page 20.)

If you know the output data format that you want, you can use the set-Format method to specify the Format and let the Processor choose an appropriate codec and renderer. Alternatively, you can specify the output format when the Processor is created by using a ProcessorModel. A ProcessorModel defines the input and output requirements for a Processor. When a ProcessorModel is passed to the appropriate Manager create method, the Manager does its best to create a Processor that meets the specified requirements.

Data Output

The getDataOutput method returns a Processor object's output as a Data-Source. This DataSource can be used as the input to another Player or Processor or as the input to a *data sink*. (For more information about data sinks, see "Media Data Storage and Transmission" on page 37.)

A Processor object's output DataSource can be of any type: PushData-Source, PushBufferDataSource, PullDataSource, or PullBufferDataSource.

Not all Processor objects output data—a Processor can render the processed data instead of outputting the data to a DataSource. A Processor that renders the media data is essentially a configurable Player.

Capture

A multimedia capturing device can act as a source for multimedia data delivery. For example, a microphone can capture raw audio input or a digital video capture board might deliver digital video from a camera. Such capture devices are abstracted as `DataSources`. For example, a device that provides timely delivery of data can be represented as a `PushDataSource`. Any type of `DataSource` can be used as a capture `DataSource`: `PushDataSource`, `PushBufferDataSource`, `PullDataSource`, or `PullBufferDataSource`.

Some devices deliver multiple data streams—for example, an audio/ video conferencing board might deliver both an audio and a video stream. The corresponding `DataSource` can contain multiple `SourceStreams` that map to the data streams provided by the device.

Media Data Storage and Transmission

A `DataSink` is used to read media data from a `DataSource` and render the media to some destination—generally a destination other than a presentation device. A particular `DataSink` might write data to a file, write data across the network, or function as an RTP broadcaster. (For more information about using a `DataSink` as an RTP broadcaster, see "Transmitting RTP Data With a Data Sink" on page 149.)

Like `Players`, `DataSink` objects are constructed through the Manager using a `DataSource`. A `DataSink` can use a `StreamWriterControl` to provide additional control over how data is written to a file. See "Writing Media Data to a File" on page 74 for more information about how `DataSink` objects are used.

Storage Controls

A `DataSink` posts a `DataSinkEvent` to report on its status. A `DataSinkEvent` can be posted with a reason code, or the `DataSink` can post one of the following `DataSinkEvent` subtypes:

- `DataSinkErrorEvent`, which indicates that an error occurred while the `DataSink` was writing data.

- `EndOfStreamEvent`, which indicates that the entire stream has successfully been written.

To respond to events posted by a DataSink, you implement the DataSinkListener interface.

Extensibility

You can extend JMF by implementing custom plug-ins, media handlers, and data sources.

Implementing Plug-Ins

By implementing one of the JMF plug-in interfaces, you can directly access and manipulate the media data associated with a Processor:

- Implementing the Demultiplexer interface enables you to control how individual tracks are extracted from a multiplexed media stream.

- Implementing the Codec interface enables you to perform the processing required to decode compressed media data, convert media data from one format to another, and encode raw media data into a compressed format.

- Implementing the Effect interface enables you to perform custom processing on the media data.

- Implementing the Multiplexer interface enables you to specify how individual tracks are combined to form a single interleaved output stream for a Processor.

- Implementing the Renderer interface enables you to control how data is processed and rendered to an output device.

Note: The JMF Plug-In API is part of the official JMF API, but JMF Players and Processors are not required to support plug-ins. Plug-ins won't work with JMF 1.0-based Players and some Processor implementations might choose not to support them. The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation fully supports the plug-in API.

Custom Codec, Effect, and Renderer plug-ins are available to a Processor through the TrackControl interface. To make a plug-in available to a default Processor or a Processor created with a ProcessorModel, you need to register it with the PlugInManager. Once you've registered your plug-in, it is included in the list of plug-ins returned by the PlugInManager get-

PlugInList method and can be accessed by the Manager when it constructs a Processor object.

Implementing MediaHandlers and DataSources

If the JMF Plug-In API doesn't provide the degree of flexibility that you need, you can directly implement several of the key JMF interfaces: Controller, Player, Processor, DataSource, and DataSink. For example, you might want to implement a high-performance Player that is optimized to present a single media format or a Controller that manages a completely different type of time-based media.

The Manager mechanism used to construct Player, Processor, DataSource, and DataSink objects enables custom implementations of these JMF interfaces to be used seamlessly with JMF. When one of the create methods is called, the Manager uses a well-defined mechanism to locate and construct the requested object. Your custom class can be selected and constructed through this mechanism once you register a unique package prefix with the PackageManager and put your class in the appropriate place in the predefined package hierarchy.

MediaHandler Construction

Players, Processors, and DataSinks are all types of MediaHandlers—they all read data from a DataSource. A MediaHandler is always constructed for a particular DataSource, which can be either identified explicitly or with a MediaLocator. When one of the create*MediaHandler* methods is called, Manager uses the content-type name obtained from the DataSource to find and create an appropriate MediaHandler object.
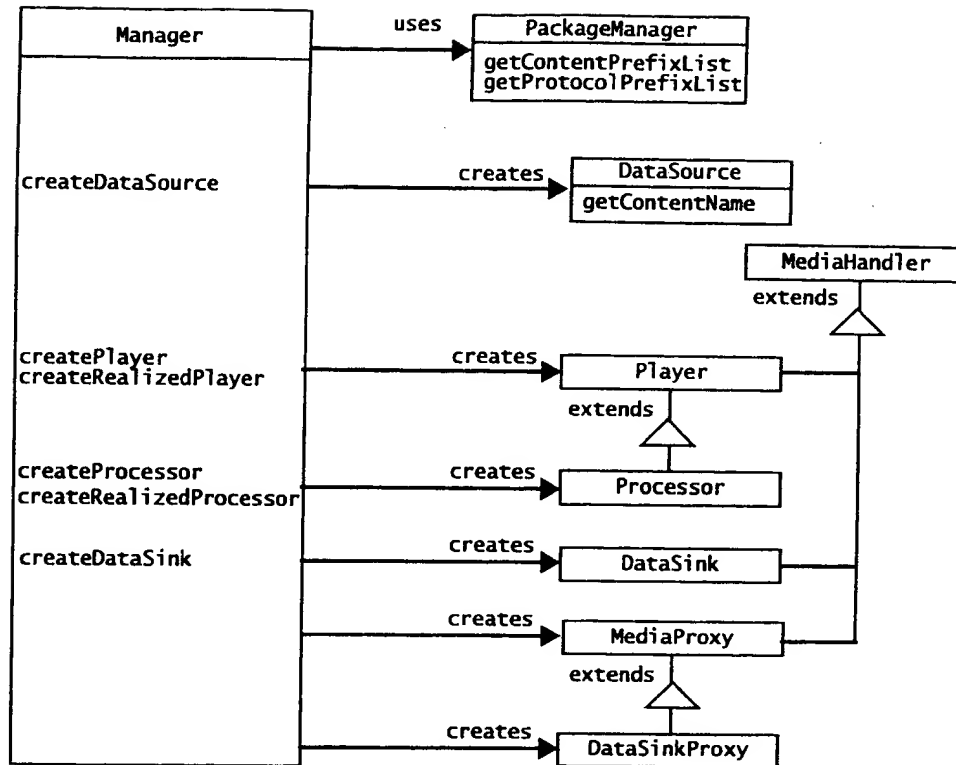
Figure 2-18: JMF media handlers.

JMF also supports another type of MediaHandler, MediaProxy. A MediaProxy processes content from one DataSource to create another. Typically, a MediaProxy reads a text configuration file that contains all of the information needed to make a connection to a server and obtain media data. To create a Player from a MediaProxy, Manager:

1. Constructs a DataSource for the protocol described by the MediaLocator

2. Uses the content-type of the DataSource to construct a MediaProxy to read the configuration file.

3. Gets a new DataSource from the MediaProxy.

4. Uses the content-type of the new DataSource to construct a Player.

The mechanism that Manager uses to locate and instantiate an appropriate MediaHandler for a particular DataSource is basically the same for all types of MediaHandlers:

- Using the list of installed content package-prefixes retrieved from PackageManager, Manager generates a search list of available MediaHandler classes.

- Manager steps through each class in the search list until it finds a class named Handler that can be constructed and to which it can attach the DataSource.

When constructing Players and Processors, Manager generates the search list of available handler classes from the list of installed *content package-prefixes* and the content-type name of the DataSource. To search for Players, Manager looks for classes of the form:

```
<content package-prefix>.media.content.<content-type>.Handler
```

To search for Processors, Manager looks for classes of the form:

```
<content package-prefix>.media.processor.<content-type>.Handler
```

If the located MediaHandler is a MediaProxy, Manager gets a new DataSource from the MediaProxy and repeats the search process.

If no appropriate MediaHandler can be found, the search process is repeated, substituting unknown for the content-type name. The unknown content type is supported by generic Players that are capable of handling a large variety of media types, often in a platform-dependent way.

Because a DataSink renders the data it reads from its DataSource to an output destination, when a DataSink is created the destination must also be taken into account. When constructing DataSinks, Manager uses the list of content package-prefixes and the protocol from the MediaLocator that identifies the destination. For each content package-prefix, Manager adds to the search list a class name of the form:

```
<content package-prefix>.media.datasink.protocol.Handler
```

If the located MediaHandler is a DataSink, Manager instantiates it, sets its DataSource and MediaLocator, and returns the resulting DataSink object. If the handler is a DataSinkProxy, Manager retrieves the content type of the proxy and generates a list of DataSink classes that support the protocol of the destination MediaLocator and the content type returned by the proxy:

```
<content package-prefix>.media.datasink.protocol.<content-type>.Handler
```

The process continues until an appropriate DataSink is located or the Manager has iterated through all of the content package-prefixes.

DataSource Construction

Manager uses the same mechanism to construct DataSources that it uses to construct MediaHandlers, except that it generates the search list of Data-Source class names from the list of installed *protocol package-prefixes*.

For each protocol package-prefix, Manager adds to the search list a class name of the form:

```
<protocol package-prefix>.media.protocol.<protocol>.DataSource
```

Manager steps through each class in the list until it finds a DataSource that it can instantiate and to which it can attach the MediaLocator.

3

Presenting Time-Based Media with JMF

To present time-based media such as audio or video with JMF, you use a Player. Playback can be controlled programmatically, or you can display a control-panel component that enables the user to control playback interactively. If you have several media streams that you want to play, you need to use a separate Player for each one. to play them in sync, you can use one of the Player objects to control the operation of the others.

A Processor is a special type of Player that can provide control over how the media data is processed before it is presented. Whether you're using a basic Player or a more advanced Processor to present media content, you use the same methods to manage playback. For information about how to control what processing is performed by a Processor, see "Processing Time-Based Media with JMF" on page 71.

The MediaPlayer bean is a Java Bean that encapsulates a JMF player to provide an easy way to present media from an applet or application. The MediaPlayer bean automatically constructs a new Player when a different media stream is selected, which makes it easier to play a series of media clips or allow the user to select which media clip to play. For information about using the MediaPlayer bean, see "Presenting Media with the Media-Player Bean" on page 66

Controlling a Player

To play a media stream, you need to construct a Player for the stream, configure the Player and prepare it to run, and then start the Player to begin playback.

Creating a Player

You create a `Player` indirectly through the media `Manager`. To display the `Player`, you get the `Player` object's components and add them to your applet's presentation space or application window.

When you need to create a new `Player`, you request it from the `Manager` by calling `createPlayer` or `createProcessor`. The `Manager` uses the media URL or `MediaLocator` that you specify to create an appropriate `Player`. A URL can only be successfully constructed if the appropriate corresponding `URL-StreamHandler` is installed. `MediaLocator` doesn't have this restriction.

Blocking Until a Player is Realized

Many of the methods that can be called on a `Player` require the `Player` to be in the *Realized* state. One way to guarantee that a `Player` is *Realized* when you call these methods is to use the `Manager` `createRealizedPlayer` method to construct the `Player`. This method provides a convenient way to create and realize a `Player` in a single step. When this method is called, it blocks until the `Player` is *Realized*. `Manager` provides an equivalent `createRealizeProcessor` method for constructing a *Realized* `Processor`.

Note: Be aware that blocking until a `Player` or `Processor` is *Realized* can produce unsatisfactory results. For example, if `createRealizedPlayer` is called in an applet, `Applet.start` and `Applet.stop` will not be able to interrupt the construction process.

Using a ProcessorModel to Create a Processor

A `Processor` can also be created using a `ProcessorModel`. The `Processor-Model` defines the input and output requirements for the `Processor` and the `Manager` does its best to create a `Processor` that meets these requirements. To create a `Processor` using a `ProcessorModel`, you call the `Manager.createRealizedProcessor` method. Example 3-1 creates a *Realized* `Processor` that can produce IMA4-encoded stereo audio tracks with a 44.1 kHz sample rate and a 16-bit sample size.

Example 3-1: Constructing a `Processor` with a `ProcessorModel`.

```
AudioFormat afs[] = new AudioFormat[1];
afs[0] = new AudioFormat("ima4", 44100, 16, 2);
Manager.createRealizedProcessor(new ProcessorModel(afs, null));
```

Since the ProcessorModel does not specify a source URL in this example, Manager implicitly finds a capture device that can capture audio and then creates a Processor that can encode that into IMA4.

Note that when you create a *Realized* Processor with a ProcessorModel you will not be able to specify processing options through the Processor object's TrackControls. For more information about specifying processing options for a Processor, see "Processing Time-Based Media with JMF" on page 71.

Displaying Media Interface Components

A Player generally has two types of user interface components, a visual component and a control-panel component. Some Player implementations can display additional components, such as volume controls and download-progress bars.

Displaying a Visual Component

A visual component is where a Player presents the visual representation of its media, if it has one. Even an audio Player might have a visual component, such as a waveform display or animated character.

To display a Player object's visual component, you:

1. Get the component by calling getVisualComponent.
2. Add it to the applet's presentation space or application window.

You can access the Player object's display properties, such as its x and y coordinates, through its visual component. The layout of the Player components is controlled through the AWT layout manager.

Displaying a Control Panel Component

A Player often has a control panel that allows the user to control the media presentation. For example, a Player might be associated with a set of buttons to start, stop, and pause the media stream, and with a slider control to adjust the volume.

Every Player provides a default control panel. To display the default control panel:

1. Call getControlPanelComponent to get the Component.

2. Add the returned Component to your applet's presentation space or application window.

If you prefer to define a custom user-interface, you can implement custom GUI Components and call the appropriate Player methods in response to user actions. If you register the custom components as ControllerListeners, you can also update them when the state of the Player changes.

Displaying a Gain-Control Component

Player implementations that support audio gain adjustments implement the GainControl interface. GainControl provides methods for adjusting the audio volume, such as setLevel and setMute. To display a GainControl Component if the Player provides one, you:

1. Call getGainControl to get the GainControl from the Player. If the Player returns null, it does not support the GainControl interface.

2. Call getControlComponent on the returned GainControl.

3. Add the returned Component to your applet's presentation space or application window.

Note that getControls does not return a Player object's GainControl. You can only access the GainControl by calling getGainControl.

Displaying Custom Control Components

Many Players have other properties that can be managed by the user. For example, a video Player might allow the user to adjust brightness and contrast, which are not managed through the Player interface. You can find out what custom controls a Player supports by calling the getControls method.

For example, you can call getControls to determine if a Player supports the CachingControl interface.

Example 3-2: Using getControls to find out what Controls are supported.

```
Control[] controls = player.getControls();
for (int i = 0; i < controls.length; i++) {
    if (controls[i] instanceof CachingControl) {
        cachingControl = (CachingControl) controls[i];
    }
}
```

Displaying a Download-Progress Component

The CachingControl interface is a special type of Control implemented by Players that can report their download progress. A CachingControl provides a default progress-bar component that is automatically updated as the download progresses. To use the default progress bar in an applet:

1. Implement the ControllerListener interface and listen for CachingControlEvents in controllerUpdate.

2. The first time you receive a CachingControlEvent:

   a. Call getCachingControl on the event to get the caching control.

   b. Call getProgressBar on the CachingControl to get the default progress bar component.

   c. Add the progress bar component to your applet's presentation space.

3. Each time you receive a CachingControlEvent, check to see if the download is complete. When getContentProgress returns the same value as getContentLength, remove the progress bar.

The Player posts a CachingControlEvent whenever the progress bar needs to be updated. If you implement your own progress bar component, you can listen for this event and update the download progress whenever CachingControlEvent is posted.

Setting the Playback Rate

The Player object's rate determines how media time changes with respect to time-base time; it defines how many units a Player object's media time advances for every unit of time-base time. The Player object's rate can be thought of as a temporal scale factor. For example, a rate of 2.0 indicates

that media time passes twice as fast as the time-base time when the Player is started.

In theory, a Player object's rate could be set to any real number, with negative rates interpreted as playing the media in reverse. However, some media formats have dependencies between frames that make it impossible or impractical to play them in reverse or at non-standard rates.

To set the rate, you call setRate and pass in the temporal scale factor as a float value. When setRate is called, the method returns the rate that is actually set, even if it has not changed. Players are only guaranteed to support a rate of 1.0.

Setting the Start Position

Setting a Player object's media time is equivalent to setting a read position within a media stream. For a media data source such as a file, the media time is bounded; the maximum media time is defined by the end of the media stream.

To set the media time you call setMediaTime and pass in a Time object that represents the time you want to set.

Frame Positioning

Some Players allow you to seek to a particular frame of a video. This enables you to easily set the start position to the beginning of particular frame without having to specify the exact media time that corresponds to that position. Players that support frame positioning implement the FramePositioningControl.

To set the frame position, you call the FramePositioningControl seek method. When you seek to a frame, the Player object's media time is set to the value that corresponds to the beginning of that frame and a MediaTimeSetEvent is posted.

Some Players can convert between media times and frame positions. You can use the FramePositioningControl mapFrameToTime and mapTimeToFrame methods to access this information, if it's available. (Players that support FramePositioningControl are not required to export this information.) Note that there is not a one-to-one correspondence between media times and frames —a frame has a duration, so several different media times might map to the same frame. (See "Getting the Media Time" on page 53 for more information.)

Preparing to Start

Most media Players cannot be started instantly. Before the Player can start, certain hardware and software conditions must be met. For example, if the Player has never been started, it might be necessary to allocate buffers in memory to store the media data. Or, if the media data resides on a network device, the Player might have to establish a network connection before it can download the data. Even if the Player has been started before, the buffers might contain data that is not valid for the current media position.

Realizing and Prefetching a Player

JMF breaks the process of preparing a Player to start into two phases, *Realizing* and *Prefetching*. *Realizing* and *Prefetching* a Player before you start it minimizes the time it takes the Player to begin presenting media when start is called and helps create a highly-responsive interactive experience for the user. Implementing the ControllerListener interface allows you to control when these operations occur.

Note: Processor introduces a third phase to the preparation process called *Configuring*. During this phase, Processor options can be selected to control how the Processor manipulates the media data. For more information, see "Selecting Track Processing Options" on page 72.

You call realize to move the Player into the *Realizing* state and begin the realization process. You call prefetch to move the Player into the *Prefetching* state and initiate the prefetching process. The realize and prefetch methods are asynchronous and return immediately. When the Player completes the requested operation, it posts a RealizeCompleteEvent or PrefetchCompleteEvent. "Player States" on page 26 describes the operations that a Player performs in each of these states.

A Player in the *Prefetched* state is prepared to start and its start-up latency cannot be further reduced. However, setting the media time through set-MediaTime might return the Player to the *Realized* state and increase its start-up latency.

Keep in mind that a *Prefetched* Player ties up system resources. Because some resources, such as sound cards, might only be usable by one program at a time, having a Player in the *Prefetched* state might prevent other Players from starting.

Determining the Start Latency

To determine how much time is required to start a Player, you can call getStartLatency. For Players that have a variable start latency, the return value of getStartLatency represents the maximum possible start latency. For some media types, getStartLatency might return LATENCY_UNKNOWN.

The start-up latency reported by getStartLatency might differ depending on the Player object's current state. For example, after a prefetch operation, the value returned by getStartLatency is typically smaller. A Controller that can be added to a Player will return a useful value once it is *Prefetched*. (For more information, see "Using a Player to Synchronize Controllers" on page 57.)

Starting and Stopping the Presentation

The Clock and Player interfaces define the methods for starting and stopping presentation.

Starting the Presentation

You typically start the presentation of media data by calling start. The start method tells the Player to begin presenting media data as soon as possible. If necessary, start prepares the Player to start by performing the realize and prefetch operations. If start is called on a *Started* Player, the only effect is that a StartEvent is posted in acknowledgment of the method call.

Clock defines a syncStart method that can be used for synchronization. See "Synchronizing Multiple Media Streams" on page 56 for more information.

To start a Player at a specific point in a media stream:

1. Specify the point in the media stream at which you want to start by calling setMediaTime.

2. Call start on the Player.

Stopping the Presentation

There are four situations in which the presentation will stop:

- When the stop method is called

- When the specified stop time is reached
- When there's no more media data to present
- When the media data is being received too slowly for acceptable playback

When a Player is stopped, its media time is frozen if the source of the media can be controlled. If the Player is presenting streamed media, it might not be possible to freeze the media time. In this case, only the receipt of the media data is stopped—the data continues to be streamed and the media time continues to advance.

When a *Stopped* Player is restarted, if the media time was frozen, presentation resumes from the stop time. If media time could not be frozen when the Player was stopped, reception of the stream resumes and playback begins with the newly-received data.

To stop a Player immediately, you call the stop method. If you call stop on a *Stopped* Player, the only effect is that a StopByRequestEvent is posted in acknowledgment of the method call.

Stopping the Presentation at a Specified Time

You can call setStopTime to indicate when a Player should stop. The Player stops when its media time passes the specified stop time. If the Player object's rate is positive, the Player stops when the media time becomes greater than or equal to the stop time. If the Player object's rate is negative, the Player stops when the media time becomes less than or equal to the stop time. The Player stops immediately if its current media time is already beyond the specified stop time.

For example, assume that a Player object's media time is 5.0 and setStopTime is called to set the stop time to 6.0. If the Player object's rate is positive, media time is increasing and the Player will stop when the media time becomes greater than or equal to 6.0. However, if the Player object's rate is negative, it is playing in reverse and the Player will stop immediately because the media time is already beyond the stop time. (For more information about Player rates, see "Setting the Playback Rate" on page 47.)

You can always call setStopTime on a *Stopped* Player. However, you can only set the stop time on a *Started* Player if the stop time is not currently

set. If the *Started* Player already has a stop time, setStopTime throws an error.

You can call getStopTime to get the currently scheduled stop time. If the clock has no scheduled stop time, getStopTime returns Clock.RESET. To remove the stop time so that the Player continues until it reaches end-of-media, call setStopTime(Clock.RESET).

Releasing Player Resources

The deallocate method tells a Player to release any exclusive resources and minimize its use of non-exclusive resources. Although buffering and memory management requirements for Players are not specified, most Players allocate buffers that are large by the standards of Java objects. A well-implemented Player releases as much internal memory as possible when deallocate is called.

The deallocate method can only be called on a *Stopped* Player. To avoid ClockStartedErrors, you should call stop before you call deallocate. Calling deallocate on a Player in the *Prefetching* or *Prefetched* state returns it to the *Realized* state. If deallocate is called while the Player is realizing, the Player posts a DeallocateEvent and returns to the *Unrealized* state. (Once a Player has been realized, it can never return to the *Unrealized* state.)

You generally call deallocate when the Player is not being used. For example, an applet should call deallocate as part of its stop method. By calling deallocate, the program can maintain references to the Player, while freeing other resources for use by the system as a whole. (JMF does not prevent a *Realized* Player that has formerly been *Prefetched* or *Started* from maintaining information that would allow it to be started up more quickly in the future.)

When you are finished with a Player (or any other Controller) and are not going to use it anymore, you should call close. The close method indicates that the Controller will no longer be used and can shut itself down. Calling close releases all of the resources that the Controller was using and causes it to cease all activity. When a Controller is closed, it posts a ControllerClosedEvent. A closed Controller cannot be reopened and invoking methods on a closed Controller might generate errors.

Querying a Player

A Player can provide information about its current parameters, including its rate, media time, and duration.

Getting the Playback Rate

To get a Player object's current rate, you call getRate. Calling getRate returns the playback rate as a float value.

Getting the Media Time

To get a Player object's current media time, you call getMediaTime. Calling getMediaTime returns the current media time as a Time object. If the Player is not presenting media data, this is the point from which media presentation will commence.

Note that there is not a one-to-one correspondence between media times and frames. Each frame is presented for a certain period of time, and the media time continues to advance during that period.

For example, imagine you have a slide show Player that displays each slide for 5 seconds—the Player essentially has a frame rate of 0.2 frames per second.
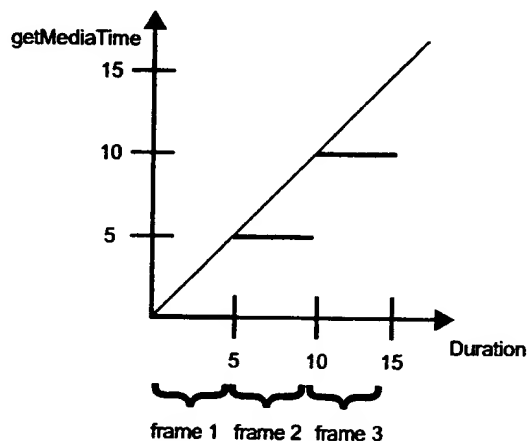
Figure 3-1: Frame duration and media time.

If you start the Player at time 0.0, while the first frame is displayed, the media time advances from 0.0 to 5.0. If you start at time 2.0, the first frame is displayed for 3 seconds, until time 5.0 is reached.

Getting the Time-Base Time

You can get a `Player` object's current time-base time by getting the `Player` object's `TimeBase` and calling `getTime`:

```
myCurrentTBTime = player1.getTimeBase().getTime();
```

When a `Player` is running, you can get the time-base time that corresponds to a particular media time by calling `mapToTimeBase`.

Getting the Duration of the Media Stream

Since programs often need to know how long a particular media stream will run, all `Controllers` implement the `Duration` interface. This interface defines a single method, `getDuration`. The duration represents the length of time that a media object would run, if played at the default rate of 1.0. A media stream's duration is only accessible through a `Player`.

If the duration can't be determined when `getDuration` is called, `DURATION_UNKNOWN` is returned. This can happen if the `Player` has not yet reached a state where the duration of the media source is available. At a later time, the duration might be available and a call to `getDuration` would return the duration value. If the media source does not have a defined duration, as in the case of a live broadcast, `getDuration` returns `DURATION_UNBOUNDED`.

Responding to Media Events

`ControllerListener` is an asynchronous interface for handling events generated by `Controller` objects. Using the `ControllerListener` interface enables you to manage the timing of potentially time-consuming `Player` operations such as prefetching.

Implementing the ControllerListener Interface

To implement the `ControllerListener` interface, you need to:

1. Implement the `ControllerListener` interface in a class.

2. Register that class as a listener by calling `addControllerListener` on the `Controller` that you want to receive events from.

When a `Controller` posts an event, it calls `controllerUpdate` on each registered listener.

Typically, `controllerUpdate` is implemented as a series of if-else statements.

Example 3-3:Implementing `controllerUpdate`.

```
if (event instanceof EventType){
...
} else if (event instanceof OtherEventType){
...
}
```

This filters out the events that you are not interested in. If you have registered as a listener with multiple `Controllers`, you also need to determine which `Controller` posted the event. `ControllerEvents` come "stamped" with a reference to their source that you can access by calling `getSource`.

When you receive events from a `Controller`, you might need to do some additional processing to ensure that the `Controller` is in the proper state before calling a control method. For example, before calling any of the methods that are restricted to *Stopped* `Players`, you should check the `Player` object's target state by calling `getTargetState`. If `start` has been called, the `Player` is considered to be in the *Started* state, though it might be posting transition events as it prepares the `Player` to present media.

Some types of `ControllerEvents` contain additional state information. For example, the `StartEvent` and `StopEvent` classes each define a method that allows you to retrieve the media time at which the event occurred.

Using ControllerAdapter

`ControllerAdapter` is a convenience class that implements `ControllerListener` and can be easily extended to respond to particular `Events`. To implement the `ControllerListener` interface using `ControllerAdapter`, you need to:

1. Subclass `ControllerAdapter` and override the event methods for the events that you're interested in.

2. Register your `ControllerAdapter` class as a listener for a particular `Controller` by calling `addControllerListener`.

When a Controller posts an event, it calls controllerUpdate on each registered listener. ControllerAdapter automatically dispatches the event to the appropriate event method, filtering out the events that you're not interested in.

For example, the following code extends a ControllerAdapter with a JDK 1.1 anonymous inner-class to create a self-contained Player that is automatically reset to the beginning of the media and deallocated when the Player reaches the end of the media.

Example 3-4: Using ControllerAdapter.

```
player.addControllerListener(new ControllerAdapter() {
    public void endOfMedia(EndOfMediaEvent e) {
        Controller controller = e.getSource();
        controller.stop();
        controller.setMediaTime(new Time(0));
        controller.deallocate();
    }
});
```

If you register a single ControllerAdapter as a listener for multiple Players, in your event method implementations you need to determine which Player generated the event. You can call getSource to determine where a ControllerEvent originated.

Synchronizing Multiple Media Streams

To synchronize the playback of multiple media streams, you can synchronize the Players by associating them with the same TimeBase. To do this, you use the getTimeBase and setTimeBase methods defined by the Clock interface. For example, you could synchronize player1 with player2 by setting player1 to use player2's time base:

```
player1.setTimeBase(player2.getTimeBase());
```

When you synchronize Players by associating them with the same Time-Base, you must still manage the control of each Player individually. Because managing synchronized Players in this way can be complicated, JMF provides a mechanism that allows a Player to assume control over any other Controller. The Player manages the states of these Controllers automatically, allowing you to interact with the entire group through a

single point of control. For more information, see See "Using a Player to Synchronize Controllers".

Using a Player to Synchronize Controllers

Synchronizing Players directly using syncStart requires that you carefully manage the states of all of the synchronized Players. You must control each one individually, listening for events and calling control methods on them as appropriate. Even with only a few Players, this quickly becomes a difficult task. Through the Player interface, JMF provides a simpler solution: a Player can be used to manage the operation of any Controller.

When you interact with a managing Player, your instructions are automatically passed along to the managed Controllers as appropriate. The managing Player takes care of the state management and synchronization for all of the other Controllers.

This mechanism is implemented through the addController and removeController methods. When you call addController on a Player, the Controller you specify is added to the list of Controllers managed by the Player. Conversely, when you call removeController, the specified Controller is removed from the list of managed Controllers.

Typically, when you need to synchronize Players or other Controllers, you should use this addController mechanism. It is simpler, faster, and less error-prone than attempting to manage synchronized Players individually.

When a Player assumes control of a Controller:

- The Controller assumes the Player object's time base.
- The Player object's duration becomes the longer of the Controller object's duration and its own. If multiple Controllers are placed under a Player object's control, the Player object's duration is set to longest duration.
- The Player object's start latency becomes the longer of the Controller object's start latency and its own. If multiple Controllers are placed under a Player object's control, the Player object's start latency is set to the longest latency.

A managing Player only posts completion events for asynchronous methods after each of its managed Controllers have posted the event. The

managing Player reposts other events generated by the Controllers as appropriate.

Adding a Controller

You use the addController method to add a Controller to the list of Controllers managed by a particular Player. To be added, a Controller must be in the *Realized* state; otherwise, a NotRealizedError is thrown. Two Players cannot be placed under control of each other. For example, if player1 is placed under the control of player2, player2 cannot be placed under the control of player1 without first removing player1 from player2's control.

Once a Controller has been added to a Player, do not call methods directly on the managed Controller. To control a managed Controller, you interact with the managing Player.

To have player2 assume control of player1, call:

```
player2.addController(player1);
```

Controlling Managed Controllers

To control the operation of a group of Controllers managed by a particular Player, you interact directly with the managing Player.

For example, to prepare all of the managed Controllers to start, call prefetch on the managing Player. Similarly, when you want to start them, call start on the managing Player. The managing Player makes sure that all of the Controllers are *Prefetched*, determines the maximum start latency among the Controllers, and calls syncStart to start them, specifying a time that takes the maximum start latency into account.

When you call a Controller method on the managing Player, the Player propagates the method call to the managed Controllers as appropriate. Before calling a Controller method on a managed Controller, the Player ensures that the Controller is in the proper state. The following table describes what happens to the managed Controllers when you call control methods on the managing Player.

Function	Stopped Player	Started Player
setMediaTime	Invokes setMediaTime on all managed Controllers.	Stops all managed Controllers, invokes setMediaTime, and restarts Controllers.
setRate	Invokes setRate on all managed Controllers. Returns the actual rate that was supported by all Controllers and set.	Stops all managed Controllers, invokes setRate, and restarts Controllers. Returns the actual rate that was supported by all Controllers and set.
start	Ensures all managed Controllers are *Prefetched* and invokes syncStart on each of them, taking into account their start latencies.	Depends on the Player implementation. Player might immediately post a StartEvent.
realize	The managing Player immediately posts a RealizeCompleteEvent. To be added, a Controller must already be realized.	The managing Player immediately posts a RealizeCompleteEvent. To be added, a Controller must already be realized.
prefetch	Invokes prefetch on all managed Controllers.	The managing Player immediately posts a PrefetchCompleteEvent, indicating that all managed Controllers are *Prefetched*.
stop	No effect.	Invokes stop on all managed Controllers.
deallocate	Invokes deallocate on all managed Controllers.	It is illegal to call deallocate on a *Started* Player.
setStopTime	Invokes setStopTime on all managed Controllers. (Player must be *Realized*.)	Invokes setStopTime on all managed Controllers. (Can only be set once on a *Started* Player.)
syncStart	Invokes syncStart on all managed Controllers.	It is illegal to call syncStart on a *Started* Player.
close	Invokes close on all managed Controllers.	It is illegal to call close on a *Started* Player.

Table 3-1: Calling control methods on a managing player.

Removing a Controller

You use the removeController method to remove a Controller from the list of controllers managed by a particular Player.

To have `player2` release control of `player1`, call:

```
player2.removeController(player1);
```

Synchronizing Players Directly

In a few situations, you might want to manage the synchronization of multiple `Player` objects yourself so that you can control the rates or media times independently. If you do this, you must:

1. Register as a listener for each synchronized `Player`.

2. Determine which `Player` object's time base is going to be used to drive the other `Player` objects and set the time base for the synchronized `Player` objects. Not all `Player` objects can assume a new time base. For example, if one of the `Player` objects you want to synchronize has a push data-source, that `Player` object's time base must be used to drive the other `Player` objects.

3. Set the rate for all of the `Players`. If a `Player` cannot support the rate you specify, it returns the rate that was used. (There is no mechanism for querying the rates that a `Player` supports.)

4. Synchronize the states of all of the `Player` objects. (For example, stop all of the players.)

5. Synchronize the operation of the `Player` objects:

   • Set the media time for each `Player`.

   • Prefetch each `Player`.

   • Determine the maximum start latency among the synchronized `Player` objects.

   • Start the `Player` objects by calling `syncStart` with a time that takes into account the maximum latency.

You must listen for transition events for all of the `Player` objects and keep track of which ones have posted events. For example, when you prefetch the `Player` objects, you need to keep track of which ones have posted `PrefetchComplete` events so that you can be sure all of them are *Prefetched* before calling `syncStart`. Similarly, when you request that the synchronized `Player` objects stop at a particular time, you need to listen

for the stop event posted by each Player to determine when all of them have actually stopped.

In some situations, you need to be careful about responding to events posted by the synchronized Player objects. To be sure of the state of all of the Player objects, you might need to wait at certain stages for all of them to reach the same state before continuing.

For example, assume that you are using one Player to drive a group of synchronized Player objects. A user interacting with that Player sets the media time to 10, starts the Player, and then changes the media time to 20. You then:

1. Pass along the first setMediaTime call to all of the synchronized Player objects.

2. Call prefetch on each Player to prepare them to start.

3. Call stop on each Player when the second set media time request is received.

4. Call setMediaTime on each Player with the new time.

5. Restart the prefetching operation.

6. When all of the Player objects have been prefetched, start them by calling syncStart, taking into account their start latencies.

In this case, just listening for PrefetchComplete events from all of the Player objects before calling syncStart isn't sufficient. You can't tell whether those events were posted in response to the first or second prefetch operation. To avoid this problem, you can block when you call stop and wait for all of the Player objects to post stop events before continuing. This guarantees that the next PrefetchComplete events you receive are the ones that you are really interested in.

Example: Playing an MPEG Movie in an Applet

The sample program PlayerApplet demonstrates how to create a Player and present an MPEG movie from within a Java applet. This is a general example that could easily be adapted to present other types of media streams.

The `Player` object's visual presentation and its controls are displayed within the applet's presentation space in the browser window. If you create a `Player` in a Java application, you are responsible for creating the window to display the `Player` object's components.

Note: While `PlayerApplet` illustrates the basic usage of a `Player`, it does not perform the error handling necessary in a real applet or application. For a more complete sample suitable for use as a template, see "JMF Applet" on page 173.

Overview of PlayerApplet

The `APPLET` tag is used to invoke `PlayerApplet` in an `HTML` file. The `WIDTH` and `HEIGHT` fields of the HTML `APPLET` tag determine the dimensions of the applet's presentation space in the browser window. The `PARAM` tag identifies the media file to be played.

Example 3-5: Invoking `PlayerApplet`.

```
<APPLET CODE=ExampleMedia.PlayerApplet
WIDTH=320 HEIGHT=300>
<PARAM NAME=FILE VALUE="sample2.mpg">
</APPLET>
```

When a user opens a web page containing `PlayerApplet`, the applet loads automatically and runs in the specified presentation space, which contains the `Player` object's visual component and default controls. The `Player` starts and plays the MPEG movie once. The user can use the default `Player` controls to stop, restart, or replay the movie. If the page containing the applet is closed while the `Player` is playing the movie, the `Player` automatically stops and frees the resources it was using.

To accomplish this, `PlayerApplet` extends `Applet` and implements the `ControllerListener` interface. `PlayerApplet` defines five methods:

- `init`—creates a `Player` for the file that was passed in through the `PARAM` tag and registers `PlayerApplet` as a controller listener so that it can observe media events posted by the `Player`. (This causes the `PlayerApplet controllerUpdate` method to be called whenever the `Player` posts an event.)
- `start`—starts the `Player` when `PlayerApplet` is started.
- `stop`—stops and deallocates the `Player` when `PlayerApplet` is stopped.
- `destroy`—closes the `Player` when `PlayerApplet` is removed.

- `controllerUpdate`—responds to `Player` events to display the `Player` object's components.

Example 3-6: `PlayerApplet`.

```java
import java.applet.*;
import java.awt.*;
import java.net.*;
import javax.media.*;

public class PlayerApplet extends Applet implements ControllerListener {
    Player player = null;
    public void init() {
        setLayout(new BorderLayout());
        String mediaFile = getParameter("FILE");
        try {
            URL mediaURL = new URL(getDocumentBase(), mediaFile);
            player = Manager.createPlayer(mediaURL);
            player.addControllerListener(this);
        }
        catch (Exception e) {
            System.err.println("Got exception "+e);
        }
    }
    public void start() {
        player.start();
    }
    public void stop() {
        player.stop();
        player.deallocate();
    }
    public void destroy() {
        player.close();
    }
    public synchronized void controllerUpdate(ControllerEvent event) {
        if (event instanceof RealizeCompleteEvent) {
            Component comp;
            if ((comp = player.getVisualComponent()) != null)
                add ("Center", comp);
            if ((comp = player.getControlPanelComponent()) != null)
                add ("South", comp);
            validate();
        }
    }
}
```

Initializing the Applet

When a Java applet starts, its `init` method is invoked automatically. You override `init` to prepare your applet to be started. `PlayerApplet` performs four tasks in `init`:

1. Retrieves the applet's FILE parameter.

2. Uses the FILE parameter to locate the media file and build a URL object that describes that media file.

3. Creates a `Player` for the media file by calling `Manager.createPlayer`.

4. Registers the applet as a controller listener with the new `Player` by calling `addControllerListener`. Registering as a listener causes the `Player-Applet controllerUpdate` method to be called automatically whenever the `Player` posts a media event. The `Player` posts media events whenever its state changes. This mechanism allows you to control the `Player` object's transitions between states and ensure that the `Player` is in a state in which it can process your requests. (For more information, see "Player States" on page 26.)

Example 3-7: Initializing `PlayerApplet`.

```java
public void init() {
    setLayout(new BorderLayout());
    // 1. Get the FILE parameter.
    String mediaFile = getParameter("FILE");
    try {
        // 2. Create a URL from the FILE parameter. The URL
        // class is defined in java.net.
        URL mediaURL = new URL(getDocumentBase(), mediaFile);
        // 3. Create a player with the URL object.
        player = Manager.createPlayer(mediaURL);
        // 4. Add PlayerApplet as a listener on the new player.
        player.addControllerListener(this);
    }
    catch (Exception e) {
        System.err.println("Got exception "+e);
    }
}
```

Controlling the Player

The `Applet` class defines `start` and `stop` methods that are called automatically when the page containing the applet is opened and closed. You override these methods to define what happens each time your applet starts and stops.

`PlayerApplet` implements `start` to start the `Player` whenever the applet is started.

Example 3-8: Starting the `Player` in `PlayerApplet`.

```
public void start() {
    player.start();
}
```

Similarly, `PlayerApplet` overrides `stop` to stop and deallocate the `Player`:

Example 3-9: Stopping the `Player` in `PlayerApplet`.

```
public void stop() {
    player.stop();
    player.deallocate();
}
```

Deallocating the `Player` releases any resources that would prevent another `Player` from being started. For example, if the `Player` uses a hardware device to present its media, `deallocate` frees that device so that other `Players` can use it.

When an applet exits, `destroy` is called to dispose of any resources created by the applet. `PlayerApplet` overrides `destroy` to close the `Player`. Closing a `Player` releases all of the resources that it's using and shuts it down permanently.

Example 3-10: Destroying the `Player` in `PlayerApplet`.

```
public void destroy() {
    player.close();
}
```

Responding to Media Events

PlayerApplet registers itself as a ControllerListener in its init method so that it receives media events from the Player. To respond to these events, PlayerApplet implements the controllerUpdate method, which is called automatically when the Player posts an event.

PlayerApplet responds to one type of event, RealizeCompleteEvent. When the Player posts a RealizeCompleteEvent, PlayerApplet displays the Player object's components.

Example 3-11: Responding to media events in PlayerApplet.

```
public synchronized void controllerUpdate(ControllerEvent event)
{
    if (event instanceof RealizeCompleteEvent) {
        Component comp;
        if ((comp = player.getVisualComponent()) != null)
            add ("Center", comp);
        if ((comp = player.getControlPanelComponent()) != null)
            add ("South", comp);
        validate();
    }
}
```

A Player object's user-interface components cannot be displayed until the Player is *Realized*; an *Unrealized* Player doesn't know enough about its media stream to provide access to its user-interface components. Player-Applet waits for the Player to post a RealizeCompleteEvent and then displays the Player object's visual component and default control panel by adding them to the applet container. Calling validate triggers the layout manager to update the display to include the new components.

Presenting Media with the MediaPlayer Bean

Using the MediaPlayer Java Bean (javax.media.bean.playerbean.Media-Player) is the simplest way to present media streams in your applets and applications. MediaPlayer encapsulates a full-featured JMF Player in a Java Bean. You can either use the MediaPlayer bean's default controls or customize its control Components.

One key advantage to using the MediaPlayer bean is that it automatically constructs a new Player when a different media stream is selected for

playback. This makes it easy to play a series of media clips or enable the user to select the media clip that they want to play.

A MediaPlayer bean has several properties that you can set, including the media source:

Property	Type	Default	Description
Show control panel	Boolean	Yes	Controls whether or not the video control panel is visible.
Loop	Boolean	Yes	Controls whether or not the media clip loops continuously.
Media location	String	N/A	The location of the media clip to be played. It can be an URL or a relative address. For example: `file:///e:/video/media/Sample1.mov` `http://webServer/media/Sample1.mov` `media/Sample1.mov`
Show caching control	Boolean	No	Controls whether or not the download-progress bar is displayed.
Fixed Aspect Ratio	Boolean	Yes	Controls whether or not the media's original fixed aspect ratio is maintained.
Volume	int	3	Controls the audio volume.

Table 3-2: Media bean properties.

To play a media clip with the MediaPlayer bean:

1. Construct an instance of MediaPlayer:

```
MediaPlayer mpl = new javax.media.bean.playerbean.MediaPlayer();
```

2. Set the location of the clip you want to play:

```
mpl.setMediaLocation(new java.lang.String("file:///E:/jvideo/media/Sample1.mov"));
```

3. Start the MediaPlayer:

```
mpl.start();
```

You can stop playback by calling stop on the MediaPlayer:

```
mpl.stop();
```

By setting up the MediaPlayer in your Applet's init method and starting the MediaPlayer in your Applet's start method, you can automatically begin media presentation when the Applet is loaded. You should call stop in the Applet's stop method so that playback halts when the Applet is stopped.

Alternatively, you can display the MediaPlayer bean's default control panel or provide custom controls to allow the user to control the media presentation. If you provide custom controls, call the appropriate MediaPlayer control and properties methods when the user interacts with the controls. For example, if you provide a custom Start button in your Applet, listen for the mouse events and call start when the user clicks on the button.

Presenting RTP Media Streams

You can present streaming media with a JMF Player constructed through the Manager using a MediaLocator that has the parameters of an RTP session. For more information about streaming media and RTP, see "Working with Real-Time Media Streams" on page 109.

When you use a MediaLocator to construct a Player for an RTP session, only the first RTP stream that's detected in the session can be presented—Manager creates a Player for the first stream that's detected in the RTP session. For information about playing multiple RTP streams from the same session, see "Receiving and Presenting RTP Media Streams" on page 129.

Note: JMF-compliant implementations are not required to support the RTP APIs in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp`. The reference implementations of JMF provided by Sun Microsystems, Inc. and IBM Corporation fully support these APIs.

Example 3-12: Creating a Player for an RTP session.

```
String url= "rtp://224.144.251.104:49150/audio/1";

MediaLocator mrl= new MediaLocator(url);

if (mrl == null) {
    System.err.println("Can't build MRL for RTP");
    return false;
}

// Create a player for this rtp session
try {
    player = Manager.createPlayer(mrl);
} catch (NoPlayerException e) {
    System.err.println("Error:" + e);
    return false;
} catch (MalformedURLException e) {
    System.err.println("Error:" + e);
    return false;
} catch (IOException e) {
    System.err.println("Error:" + e);
    return false;
}
```

When data is detected on the session, the Player posts a RealizeComple-teEvent. By listening for this event, you can determine whether or not any data has arrived and if the Player is capable of presenting any data. Once the Player posts this event, you can retrieve its visual and control components.

Listening for RTP Format Changes

When a Player posts a FormatChangeEvent, it can indicate that a payload change has occurred. Player objects constructed with a MediaLocator automatically process payload changes. In most cases, this processing involves constructing a new Player to handle the new format. Programs that present RTP media streams need to listen for FormatChangeEvents so that they can respond if a new Player is created.

When a FormatChangeEvent is posted, check whether or not the Player object's control and visual components have changed. If they have, a new Player has been constructed and you need to remove references to the old Player object's components and get the new Player object's components.

Working with Real-Time Media Streams

To send or receive a live media broadcast or conduct a video conference over the Internet or Intranet, you need to be able to receive and transmit media streams in real-time. This chapter introduces streaming media concepts and describes the Real-time Transport Protocol JMF uses for receiving and transmitting media streams across the network.

Streaming Media

When media content is streamed to a client in real-time, the client can begin to play the stream without having to wait for the complete stream to download. In fact, the stream might not even have a predefined duration—downloading the entire stream before playing it would be impossible. The term *streaming media* is often used to refer to both this technique of delivering content over the network in real-time and the real-time media content that's delivered.

Streaming media is everywhere you look on the web—live radio and television broadcasts and webcast concerts and events are being offered by a rapidly growing number of web portals, and it's now possible to conduct audio and video conferences over the Internet. By enabling the delivery of dynamic, interactive media content across the network, streaming media is changing the way people communicate and access information.

Protocols for Streaming Media

Transmitting media data across the net in real-time requires high network throughput. It's easier to compensate for lost data than to compensate for

large delays in receiving the data. This is very different from accessing static data such as a file, where the most important thing is that all of the data arrive at its destination. Consequently, the protocols used for static data don't work well for streaming media.

The HTTP and FTP protocols are based on the Transmission Control Protocol (TCP). TCP is a transport-layer protocol[1] designed for reliable data communications on low-bandwidth, high-error-rate networks. When a packet is lost or corrupted, it's retransmitted. The overhead of guaranteeing reliable data transfer slows the overall transmission rate.

For this reason, underlying protocols other than TCP are typically used for streaming media. One that's commonly used is the User Datagram Protocol (UDP). UDP is an unreliable protocol; it does not guarantee that each packet will reach its destination. There's also no guarantee that the packets will arrive in the order that they were sent. The receiver has to be able to compensate for lost data, duplicate packets, and packets that arrive out of order.

Like TCP, UDP is a general transport-layer protocol—a lower-level networking protocol on top of which more application-specific protocols are built. The Internet standard for transporting real-time data such as audio and video is the Real-Time Transport Protocol (RTP).

RTP is defined in IETF RFC 1889, a product of the AVT working group of the Internet Engineering Task Force (IETF).

Real-Time Transport Protocol

RTP provides end-to-end network delivery services for the transmission of real-time data. RTP is network and transport-protocol independent, though it is often used over UDP.

---

[1] In the seven layer ISO/OSI data communications model, the transport layer is level four. For more information about the ISO/OSI model, see *Understanding OSI.* Larmouth, John. International Thompson Computer Press, 1996. ISBN 1850321760.
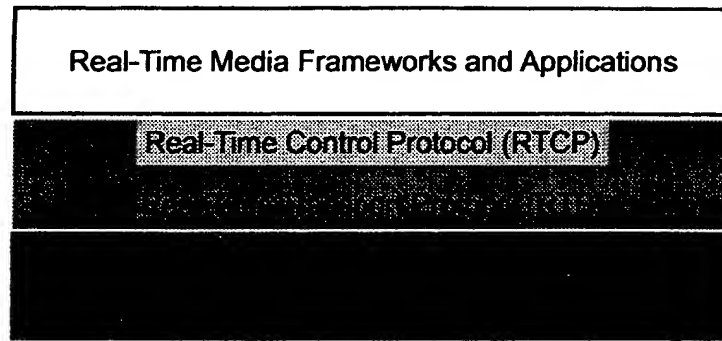
Figure 7-1: RTP architecture.

RTP can be used over both unicast and multicast network services. Over a *unicast* network service, separate copies of the data are sent from the source to each destination. Over a *multicast* network service, the data is sent from the source only once and the network is responsible for transmitting the data to multiple locations. Multicasting is more efficient for many multimedia applications, such as video conferences. The standard Internet Protocol (IP) supports multicasting.

RTP Services

RTP enables you to identify the type of data being transmitted, determine what order the packets of data should be presented in, and synchronize media streams from different sources.

RTP data packets are not guaranteed to arrive in the order that they were sent—in fact, they're not guaranteed to arrive at all. It's up to the receiver to reconstruct the sender's packet sequence and detect lost packets using the information provided in the packet header.

While RTP does not provide any mechanism to ensure timely delivery or provide other quality of service guarantees, it is augmented by a control protocol (RTCP) that enables you to monitor the quality of the data distribution. RTCP also provides control and identification mechanisms for RTP transmissions.

If quality of service is essential for a particular application, RTP can be used over a resource reservation protocol that provides connection-oriented services.

RTP Architecture

An RTP *session* is an association among a set of applications communicating with RTP. A session is identified by a network address and a pair of ports. One port is used for the media data and the other is used for control (RTCP) data.

A *participant* is a single machine, host, or user participating in the session. Participation in a session can consist of passive reception of data (receiver), active transmission of data (sender), or both.

Each media type is transmitted in a different session. For example, if both audio and video are used in a conference, one session is used to transmit the audio data and a separate session is used to transmit the video data. This enables participants to choose which media types they want to receive—for example, someone who has a low-bandwidth network connection might only want to receive the audio portion of a conference.

Data Packets

The media data for a session is transmitted as a series of packets. A series of data packets that originate from a particular source is referred to as an *RTP stream*. Each RTP data packet in a stream contains two parts, a structured header and the actual data (the packet's *payload*).
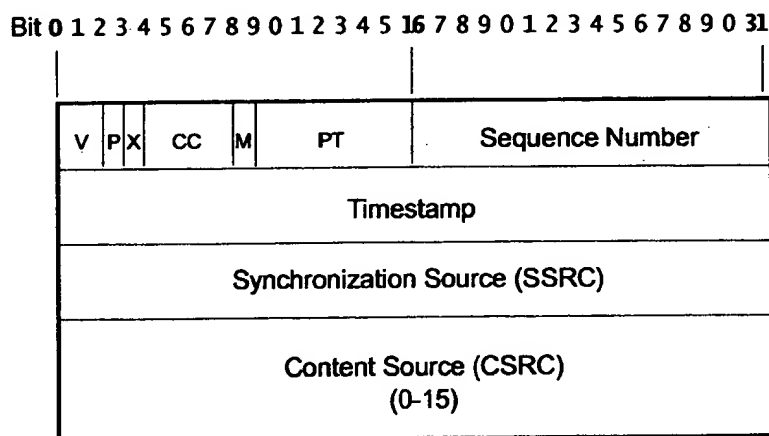
Figure 7-2: RTP data-packet header format.

The header of an RTP data packet contains:

- **The RTP version number (V):** 2 bits. The version defined by the current specification is 2.

- **Padding (P)**: 1 bit. If the padding bit is set, there are one or more bytes at the end of the packet that are not part of the payload. The very last byte in the packet indicates the number of bytes of padding. The padding is used by some encryption algorithms.

- **Extension (X)**: 1 bit. If the extension bit is set, the fixed header is followed by one header extension. This extension mechanism enables implementations to add information to the RTP Header.

- **CSRC Count (CC)**: 4 bits. The number of CSRC identifiers that follow the fixed header. If the CSRC count is zero, the synchronization source is the source of the payload.

- **Marker (M)**: 1 bit. A marker bit defined by the particular media profile.

- **Payload Type (PT)**: 7 bits. An index into a media profile table that describes the payload format. The payload mappings for audio and video are specified in RFC 1890.

- **Sequence Number**: 16 bits. A unique packet number that identifies this packet's position in the sequence of packets. The packet number is incremented by one for each packet sent.

- **Timestamp**: 32 bits. Reflects the sampling instant of the first byte in the payload. Several consecutive packets can have the same timestamp if they are logically generated at the same time—for example, if they are all part of the same video frame.

- **SSRC**: 32 bits. Identifies the synchronization source. If the CSRC count is zero, the payload source is the synchronization source. If the CSRC count is nonzero, the SSRC identifies the mixer..

- **CSRC**: 32 bits each. Identifies the contributing sources for the payload. The number of contributing sources is indicated by the CSRC count field; there can be up to 16 contributing sources. If there are multiple contributing sources, the payload is the mixed data from those sources.

Control Packets

In addition to the media data for a session, control data (RTCP) packets are sent periodically to all of the participants in the session. RTCP packets can contain information about the quality of service for the session participants, information about the source of the media being transmitted on the data port, and statistics pertaining to the data that has been transmitted so far.

There are several types of RTCP packets:

- Sender Report
- Receiver Report
- Source Description
- Bye
- Application-specific

RTCP packets are "stackable" and are sent as a compound packet that contains at least two packets, a report packet and a source description packet.

All participants in a session send RTCP packets. A participant that has recently sent data packets issues a *sender report*. The sender report (SR) contains the total number of packets and bytes sent as well as information that can be used to synchronize media streams from different sessions.

Session participants periodically issue *receiver reports* for all of the sources from which they are receiving data packets. A receiver report (RR) contains information about the number of packets lost, the highest sequence number received, and a timestamp that can be used to estimate the round-trip delay between a sender and the receiver.

The first packet in a compound RTCP packet has to be a report packet, even if no data has been sent or received—in which case, an empty receiver report is sent.

All compound RTCP packets must include a source description (SDES) element that contains the canonical name (CNAME) that identifies the source. Additional information might be included in the source description, such as the source's name, email address, phone number, geographic location, application name, or a message describing the current state of the source.

When a source is no longer active, it sends an RTCP BYE packet. The BYE notice can include the reason that the source is leaving the session.

RTCP APP packets provide a mechanism for applications to define and send custom information via the RTP control port.

RTP Applications

RTP applications are often divided into those that need to be able to receive data from the network (RTP Clients) and those that need to be able

to transmit data across the network (RTP Servers). Some applications do both—for example, conferencing applications capture and transmit data at the same time that they're receiving data from the network.

Receiving Media Streams From the Network

Being able to receive RTP streams is necessary for several types of applications. For example:

- Conferencing applications need to be able to receive a media stream from an RTP session and render it on the console.

- A telephone answering machine application needs to be able to receive a media stream from an RTP session and store it in a file.

- An application that records a conversation or conference must be able to receive a media stream from an RTP session and both render it on the console and store it in a file.

Transmitting Media Streams Across the Network

RTP server applications transmit captured or stored media streams across the network.

For example, in a conferencing application, a media stream might be captured from a video camera and sent out on one or more RTP sessions. The media streams might be encoded in multiple media formats and sent out on several RTP sessions for conferencing with heterogeneous receivers. Multiparty conferencing could be implemented without IP multicast by using multiple unicast RTP sessions.

References

The RTP specification is a product of the Audio Video Transport (AVT) working group of the Internet Engineering Task Force (IETF). For additional information about the IETF, see http://www.ietf.org. The AVT working group charter and proceedings are available at http://www.ietf.org/html.charters/avt-charter.html.

IETF RFC 1889, RTP: A Transport Protocol for Real Time Applications
Current revision: http://www.ietf.org.internet-drafts/draft-ietf-avt-rtp-new-04.txt

IETF RFC 1890: RTP Profile for Audio and Video Conferences with Minimal Control
Current revision: `http://www.ietf.org.internet-drafts/draft-ietf-avt-profile-new-06.txt`

Note: These RFCs are undergoing revisions in preparation for advancement from Proposed Standard to Draft Standard and the URLs listed here are for the Internet Drafts of the revisions available at the time of publication.

In addition to these RFCs, separate payload specification documents define how particular payloads are to be carried in RTP. For a list of all of the RTP-related specifications, see the AVT working group charter at: `http://www.ietf.org/html.charters/avt-charter.html`.

Understanding the JMF RTP API

JMF enables the playback and transmission of RTP streams through the APIs defined in the `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp` packages. JMF can be extended to support additional RTP-specific formats and dynamic payloads through the standard JMF plug-in mechanism.

Note: JMF-compliant implementations are not required to support the RTP APIs in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp`. The reference implementations of JMF provided by Sun Microsystems, Inc. and IBM Corporation fully support these APIs.

You can play incoming RTP streams locally, save them to a file, or both.
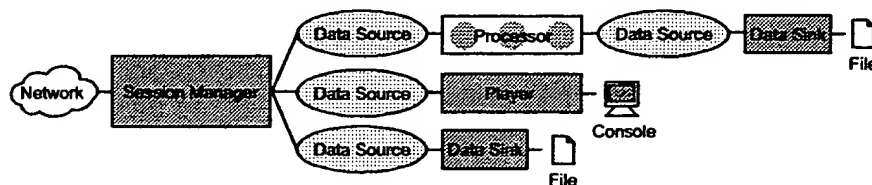
Figure 8-1: RTP reception.

For example, the RTP APIs could be used to implement a telephony application that answers calls and records messages like an answering machine.

Similarly, you can use the RTP APIs to transmit captured or stored media streams across the network. Outgoing RTP streams can originate from a file or a capture device. The outgoing streams can also be played locally, saved to a file, or both.
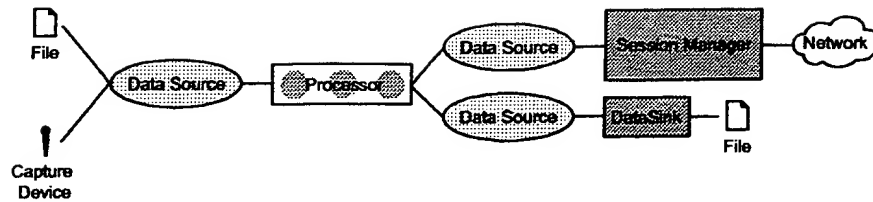
Figure 8-2: RTP transmission.

For example, you could implement a video conferencing application that captures live audio and video and transmits it across the network using a separate RTP session for each media type.

Similarly, you might record a conference for later broadcast or use a prerecorded audio stream as "hold music" in a conferencing application.

RTP Architecture

The JMF RTP APIs are designed to work seamlessly with the capture, presentation, and processing capabilities of JMF. Players and processors are used to present and manipulate RTP media streams just like any other media content. You can transmit media streams that have been captured from a local capture device using a capture DataSource or that have been stored to a file using a DataSink. Similarly, JMF can be extended to support additional RTP formats and payloads through the standard plug-in mechanism.
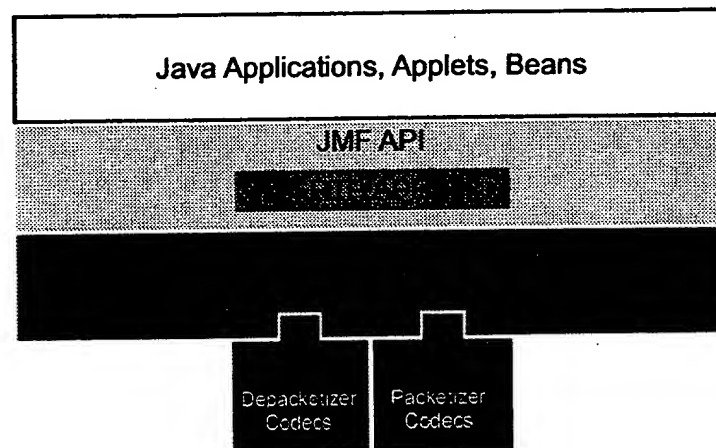
Figure 8-3: High-level JMF RTP architecture.

Session Manager

In JMF, a SessionManager is used to coordinate an RTP session. The session manager keeps track of the session participants and the streams that are being transmitted.

The session manager maintains the state of the session as viewed from the local participant. In effect, a session manager is a local representation of a distributed entity, the RTP session. The session manager also handles the RTCP control channel, and supports RTCP for both senders and receivers.

The SessionManager interface defines methods that enable an application to initialize and start participating in a session, remove individual streams created by the application, and close the entire session.

Session Statistics

The session manager maintains statistics on all of the RTP and RTCP packets sent and received in the session. Statistics are tracked for the entire session on a per-stream basis. The session manager provides access to global reception and transmission statistics:

- **GlobalReceptionStats**: Maintains global reception statistics for the session.

- **GlobalTransmissionStats**: Maintains cumulative transmission statistics for all local senders.

Statistics for a particular recipient or outgoing stream are available from the stream:

- **ReceptionStats**: Maintains source reception statistics for an individual participant.

- **TransmissionStats**: Maintains transmission statistics for an individual send stream.

Session Participants

The session manager keeps track of all of the participants in a session. Each participant is represented by an instance of a class that implements the Participant interface. SessionManagers create a Participant whenever an RTCP packet arrives that contains a source description (SDES) with a canonical name (CNAME) that has not been seen before in the session (or has timed-out since its last use). Participants can be passive (sending

control packets only) or active (also sending one or more RTP data streams).

There is exactly one *local participant* that represents the local client/server participant. A local participant indicates that it will begin sending RTCP control messages or data and maintain state on incoming data and control messages by starting a session.

A participant can own more than one stream, each of which is identified by the synchronization source identifier (SSRC) used by the source of the stream.

Session Streams

The `SessionManager` maintains an `RTPStream` object for each stream of RTP data packets in the session. There are two types of RTP streams:

- `ReceiveStream` represents a stream that's being received from a remote participant.
- `SendStream` represents a stream of data coming from the `Processor` or input `DataSource` that is being sent over the network.

A `ReceiveStream` is constructed automatically whenever the session manager detects a new source of RTP data. To create a new `SendStream`, you call the `SessionManager createSendStream` method.

RTP Events

Several RTP-specific events are defined in `javax.media.rtp.event`. These events are used to report on the state of the RTP session and streams.

```
┌─────────────┐
│ MediaEvent  │
└─────────────┘
       △
┌─────────────┐
│ RTPEvent    │
└─────────────┘
       △
       │    ┌──────────────────┐
       │    │ ReceiveStreamEvent│
       │    └──────────────────┘
       │           △
       │           │  ┌──────────────────────────┐
       │           ├──│ ActiveReceiveStreamEvent  │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           ├──│ ApplicationEvent          │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           ├──│ InactiveReceiveStreamEvent│
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           ├──│ NewReceiveStreamEvent     │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           ├──│ RemotePayloadChangeEvent  │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           ├──│ StreamMappedEvent         │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           └──│ TimeoutEvent              │
       │              └──────────────────────────┘
       │    ┌─────────────┐
       ├────│ RemoteEvent │
       │    └─────────────┘
       │           △
       │           │  ┌──────────────────────────┐
       │           ├──│ ReceiverReportEvent       │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           ├──│ SenderReportEvent         │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           └──│ RemoteCollisionEvent      │
       │              └──────────────────────────┘
       │    ┌────────────────┐
       ├────│ SendStreamEvent│
       │    └────────────────┘
       │           △
       │           │  ┌──────────────────────────┐
       │           ├──│ ActiveSendStreamEvent     │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           ├──│ InactiveSendStreamEvent   │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           ├──│ NewSendStreamEvent        │
       │           │  └──────────────────────────┘
       │           │  ┌──────────────────────────┐
       │           └──│ LocalPayloadChangeEvent   │
       │              └──────────────────────────┘
       │    ┌─────────────┐
       └────│ SessionEvent│
            └─────────────┘
                   △
                   │  ┌──────────────────────────┐
                   ├──│ LocalCollisionEvent       │
                   │  └──────────────────────────┘
                   │  ┌──────────────────────────┐
                   └──│ NewParticipantEvent       │
                      └──────────────────────────┘
```

Figure 8-4: RTP events.

To receive notification of RTP events, you implement the appropriate RTP listener and register it with the session manager:

- **SessionListener**: Receives notification of changes in the state of the session.

- **SendStreamListener**: Receives notification of changes in the state of an RTP stream that's being transmitted.

- **ReceiveStreamListener**: Receives notification of changes in the state of an RTP stream that's being received.

- **RemoteListener**: Receives notification of events or RTP control messages received from a remote participant.

Session Listener

You can implement SessionListener to receive notification about events that pertain to the RTP session as a whole, such as the addition of new participants.

There are two types of session-wide events:

- **NewParticipantEvent**: Indicates that a new participant has joined the session.
- **LocalCollisionEvent**: Indicates that the participant's synchronization source is already in use.

Send Stream Listener

You can implement SendStreamListener to receive notification whenever:

- New send streams are created by the local participant.
- The transfer of data from the DataSource used to create the send stream has started or stopped.
- The send stream's format or payload changes.

There are five types of events associated with a SendStream:

- **NewSendStreamEvent**: Indicates that a new send stream has been created by the local participant.

- **ActiveSendStreamEvent**: Indicates that the transfer of data from the DataSource used to create the send stream has started.

- **InactiveSendStreamEvent**: Indicates that the transfer of data from the DataSource used to create the send stream has stopped.

- **LocalPayloadChangeEvent**: Indicates that the stream's format or payload has changed.

- **StreamClosedEvent**: Indicates that the stream has been closed.

Receive Stream Listener

You can implement ReceiveStreamListener to receive notification whenever:

- New receive streams are created.
- The transfer of data starts or stops.
- The data transfer times out.
- A previously orphaned ReceiveStream has been associated with a Participant.
- An RTCP APP packet is received.
- The receive stream's format or payload changes.

You can also use this interface to get a handle on the stream and access the RTP DataSource so that you can create a MediaHandler.

There are seven types of events associated with a ReceiveStream:

- **NewReceiveStreamEvent**: Indicates that the session manager has created a new receive stream for a newly-detected source.
- **ActiveReceiveStreamEvent**: Indicates that the transfer of data has started.
- **InactiveReceiveStreamEvent**: Indicates that the transfer of data has stopped.
- **TimeoutEvent**: Indicates that the data transfer has timed out.
- **RemotePayloadChangeEvent**: Indicates that the format or payload of the receive stream has changed.
- **StreamMappedEvent**: Indicates that a previously orphaned receive stream has been associated with a participant.
- **ApplicationEvent**: Indicates that an RTCP APP packet has been received.

Remote Listener

You can implement RemoteListener to receive notification of events or RTP control messages received from a remote participants. You might want to implement RemoteListener in an application used to monitor the

session—it enables you to receive RTCP reports and monitor the quality of the session reception without having to receive data or information on each stream.

There are three types of events associated with a remote participant:

- **ReceiverReportEvent**: Indicates that an RTP receiver report has been received.

- **SenderReportEvent**: Indicates that an RTP sender report has been received.

- **RemoteCollisionEvent**: Indicates that two remote participants are using the same synchronization source ID (SSRC).

RTP Data

The streams within an RTP session are represented by RTPStream objects. There are two types of RTPStreams: ReceiveStream and SendStream. Each RTP stream has a buffer data source associated with it. For ReceiveStreams, this DataSource is always a PushBufferDataSource.

The session manager automatically constructs new receive streams as it detects additional streams arriving from remote participants. You construct new send streams by calling createSendStream on the session manager.

Data Handlers

The JMF RTP APIs are designed to be transport-protocol independent. A custom RTP data handler can be created to enable JMF to work over a specific transport protocol. The data handler is a DataSource that can be used as the media source for a Player.

The abstract class RTPPushDataSource defines the basic elements of a JMF RTP data handler. A data handler has both an input data stream (PushSourceStream) and an output data stream (OuputDataStream). A data handler can be used for either the data channel or the control channel of an RTP session. If it is used for the data channel, the data handler implements the DataChannel interface.

An RTPSocket is an RTPPushDataSource has both a data and control channel. Each channel has an input and output stream to stream data to and from the underlying network. An RTPSocket can export RTPControls to add dynamic payload information to the session manager.

Because a custom RTPSocket can be used to construct a Player through the Manager, JMF defines the name and location for custom RTPSocket implementations:

```
<protocol package-prefix>.media.protocol.rtpraw.DataSource
```

RTP Data Formats

All RTP-specific data uses an RTP-specific format encoding as defined in the AudioFormat and VideoFormat classes. For example, gsm RTP encapsulated packets have the encoding set to AudioFormat.GSM_RTP, while jpeg-encoded video formats have the encoding set to VideoFormat.JPEG_RTP.

AudioFormat defines four standard RTP-specific encoding strings:

```
public static final String ULAW_RTP = "JAUDIO_G711_ULAW/rtp";
public static final String DVI_RTP = "dvi/rtp";
public static final String G723_RTP = "g723/rtp";
public static final String GSM_RTP = "gsm/rtp";
```

VideoFormat defines three standard RTP-specific encoding strings:

```
public static final String JPEG_RTP = "jpeg/rtp";
public static final String H261_RTP = "h261/rtp";
public static final String H263_RTP = "h263/rtp";
```

RTP Controls

The RTP API defines one RTP-specific control, RTPControl. RTPControl is typically implemented by RTP-specific DataSources. It provides a mechanism to add a mapping between a dynamic payload and a Format. RTPControl also provides methods for accessing session statistics and getting the current payload Format.

SessionManager also extends the Controls interface, enabling a session manager to export additional Controls through the getControl and get-Controls methods. For example, the session manager can export a Buffer-Control to enable you to specify the buffer length and threshold.

Reception

The presentation of an incoming RTP stream is handled by a Player. To receive and present a single stream from an RTP session, you can use a

MediaLocator that describes the session to construct a Player. A media locator for an RTP session is of the form:

```
rtp://address:port[:ssrc]/content-type/[ttl]
```

The Player is constructed and connected to the first stream in the session.

If there are multiple streams in the session that you want to present, you need to use a session manager. You can receive notification from the session manager whenever a stream is added to the session and construct a Player for each new stream. Using a session manager also enables you to directly monitor and control the session.

Transmission

A session manager can also be used to initialize and control a session so that you can stream data across the network. The data to be streamed is acquired from a Processor.

For example, to create a send stream to transmit data from a live capture source, you would:

1. Create, initialize, and start a SessionManager for the session.

2. Construct a Processor using the appropriate capture DataSource.

3. Set the output format of the Processor to an RTP-specific format. An appropriate RTP packetizer codec must be available for the data format you want to transmit.

4. Retrieve the output DataSource from the Processor.

5. Call createSendStream on the session manager and pass in the Data-Source.

You control the transmission through the SendStream start and stop methods.

When it is first started, the SessionManager behaves as a receiver (sends out RTCP receiver reports). As soon as a SendStream is created, it begins to send out RTCP sender reports and behaves as a sender host as long as one or more send streams exist. If all SendStreams are closed (not just stopped), the session manager reverts to being a passive receiver.

Extensibility

Like the other parts of JMF, the RTP capabilities can be enhanced and extended. The RTP APIs support a basic set of RTP formats and payloads. Advanced developers and technology providers can implement JMF plug-ins to support dynamic payloads and additional RTP formats.

Implementing Custom Packetizers and Depacketizers

To implement a custom packetizer or depacketizer, you implement the JMF Codec interface. (For general information about JMF plug-ins, see "Implementing JMF Plug-Ins" on page 85.)

Receiving and Presenting RTP Media Streams

JMF Players and Processors provide the presentation, capture, and data conversion mechanisms for RTP streams.
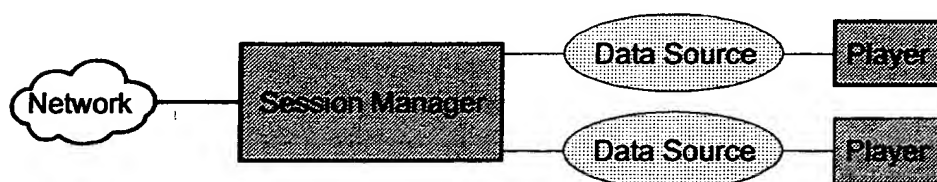
Figure 9-1: RTP reception data flow.

A separate player is used for each stream received by the session manager. You construct a Player for an RTP stream through the standard Manager createPlayer mechanism. You can either:

- Use a MediaLocator that has the parameters of the RTP session and construct a Player by calling Manager.createPlayer(MediaLocator)

- Construct a Player for a particular ReceiveStream by retrieving the DataSource from the stream and passing it to Manager.createPlayer(DataSource).

If you use a MediaLocator to construct a Player, you can only present the first RTP stream that's detected in the session. If you want to play back multiple RTP streams in a session, you need to use the SessionManager directly and construct a Player for each ReceiveStream.

Creating a Player for an RTP Session

When you use a MediaLocator to construct a Player for an RTP session, the Manager creates a Player for the first stream detected in the session. This Player posts a RealizeCompleteEvent once data has been detected in the session.

By listening for the RealizeCompleteEvent, you can determine whether or not any data has arrived and if the Player is capable of presenting any data. Once the Player posts this event, you can retrieve its visual and control components.

Note: Because a Player for an RTP media stream doesn't finish realizing until data is detected in the session, you shouldn't try to use Manager.createRealizedPlayer to construct a Player for an RTP media stream. No Player would be returned until data arrives and if no data is detected, attempting to create a *Realized* Player would block indefinitely.

A Player can export one RTP-specific control, RTPControl, which provides overall session statistics and can be used for registering dynamic payloads with the SessionManager.

Example 9-1: Creating a Player for an RTP session (1 of 2)

```java
String url= "rtp://224.144.251.104:49150/audio/1";

MediaLocator mrl= new MediaLocator(url);

if (mrl == null) {
    System.err.println("Can't build MRL for RTP");
    return false;
}

// Create a player for this rtp session
try {
    player = Manager.createPlayer(mrl);
} catch (NoPlayerException e) {
    System.err.println("Error:" + e);
    return false;
} catch (MalformedURLException e) {
    System.err.println("Error:" + e);
    return false;
} catch (IOException e) {
    System.err.println("Error:" + e);
    return false;
}

if (player != null) {
    if (this.player == null) {
```

Example 9-1: Creating a Player for an RTP session (2 of 2)

```
                        this.player = player;
                        player.addControllerListener(this);
                        player.realize();
            }
        }
```

Listening for Format Changes

When a Player posts a FormatChangeEvent, it might indicate that a payload change has occurred. Players constructed with a MediaLocator automatically process payload changes. In most cases, this processing involves constructing a new Player to handle the new format. Applications that present RTP media streams need to listen for FormatChangeEvents so that they can respond if a new Player is created.

When a FormatChangeEvent is posted, check whether or not the Player object's control and visual components have changed. If they have, a new Player has been constructed and you need to remove references to the old Player object's components and get the new Player object's components.

Example 9-2: Listening for RTP format changes (1 of 2)

```
public synchronized void controllerUpdate(ControllerEvent ce) {
    if (ce instanceof FormatChangeEvent) {
        Dimension vSize = new Dimension(320,0);
        Component oldVisualComp = visualComp;

        if ((visualComp = player.getVisualComponent()) != null) {
            if (oldVisualComp != visualComp) {
                if (oldVisualComp != null) {
                    oldVisualComp.remove(zoomMenu);
                }

                framePanel.remove(oldVisualComp);

                vSize = visualComp.getPreferredSize();
                vSize.width = (int)(vSize.width * defaultScale);
                vSize.height = (int)(vSize.height * defaultScale);

                framePanel.add(visualComp);

                visualComp.setBounds(0,
                                     0,
                                     vSize.width,
                                     vSize.height);
                addPopupMenu(visualComp);
            }
```

Example 9-2: Listening for RTP format changes (2 of 2)

```
        }
        Component oldComp = controlComp;

        controlComp = player.getControlPanelComponent();

        if (controlComp != null)
        {
            if (oldComp != controlComp)
            {
                framePanel.remove(oldComp);
                framePanel.add(controlComp);

                if (controlComp != null) {
                    int prefHeight = controlComp
                                        .getPreferredSize()
                                        .height;

                    controlComp.setBounds(0,
                                        vSize.height,
                                        vSize.width,
                                        prefHeight);
                }
            }
        }
    }
}
```

Creating an RTP Player for Each New Receive Stream

To play all of the ReceiveStreams in a session, you need to create a separate Player for each stream. When a new stream is created, the session manager posts a NewReceiveStreamEvent. Generally, you register as a ReceiveStreamListener and construct a Player for each new ReceiveStream. To construct the Player, you retrieve the DataSource from the ReceiveStream and pass it to Manager.createPlayer.

To create a Player for each new receive stream in a session:

1. Set up the RTP session:
   a. Create a SessionManager. For example, construct an instance of com.sun.media.rtp.RTPSessionMgr. (RTPSessionMgr is an implementation of SessionManager provided with the JMF reference implementation.)

b. Call `RTPSessionMgr addReceiveStreamListener` to register as a listener.

c. Initialize the RTP session by calling `RTPSessionMgr initSession`.

d. Start the RTP session by calling `RTPSessionMgr startSession`.

Example 9-3: Setting up an RTP session (1 of 2)

```
public SessionManager createManager(String address,
                                    int port,
                                    int ttl,
                                    boolean listener,
                                    boolean sendlistener)
{
    mgr = (SessionManager)new com.sun.media.rtp.RTPSessionMgr();

    if (mgr == null) return null;

    mgr.addFormat(new AudioFormat(AudioFormat.DVI_RTP,
                                  44100,
                                  4,
                                  1),
                  18);

    if (listener) mgr.addReceiveStreamListener(this);
    if (sendlistener) new RTPSendStreamWindow(mgr);

    // ask session mgr to generate the local participant's CNAME
    String cname = mgr.generateCNAME();
    String username = null;

    try {
        username = System.getProperty("user.name");
    } catch (SecurityException e){
        username = "jmf-user";
    }

    // create our local Session Address
    SessionAddress localaddr = new SessionAddress();

    try{
        InetAddress destaddr = InetAddress.getByName(address);

        SessionAddress sessaddr = new SessionAddress(destaddr,
                                                     port,
                                                     destaddr,
                                                     port + 1);
        SourceDescription[] userdesclist= new SourceDescription[]
        {
            new SourceDescription(SourceDescription
                    .SOURCE_DESC_EMAIL,
                    "jmf-user@sun.com",
```

Example 9-3: Setting up an RTP session (2 of 2)

```
                                  1,
                                  false),

          new SourceDescription(SourceDescription
                              .SOURCE_DESC_CNAME,
                              cname,
                              1,
                              false),

          new SourceDescription(SourceDescription
                              .SOURCE_DESC_TOOL,
                              "JMF RTP Player v2.0",
                              1,
                              false)
          };

          mgr.initSession(localaddr,
                          userdesclist,
                          0.05,
                          0.25);

          mgr.startSession(sessaddr,ttl,null);
      } catch (Exception e) {
          System.err.println(e.getMessage());
          return null;
      }

      return mgr;
  }
```

2. In your `ReceiveStreamListener` update method, watch for `NewReceiveStreamEvent`, which indicates that a new data stream has been detected.

3. When a `NewReceiveStreamEvent` is detected, retrieve the `ReceiveStream` from the `NewReceiveStreamEvent` by calling `getReceiveStream`.

4. Retrieve the RTP `DataSource` from the `ReceiveStream` by calling `getDataSource`. This is a `PushBufferDataSource` with an RTP-specific Format. For example, the encoding for a DVI audio player will be DVI_RTP.

5. Pass the `DataSource` to `Manager.createPlayer` to construct a `Player`. For the `Player` to be successfully constructed, the necessary plug-ins for decoding and depacketizing the RTP-formatted data must be available. (For more information, see "Creating Custom Packetizers and Depacketizers" on page 167).

Example 9-4: Listening for `NewReceiveStreamEvents`

```
    public void update( ReceiveStreamEvent event)
    {
        Player newplayer = null;
        RTPPlayerWindow playerWindow = null;

        // find the sourceRTPSM for this event
        SessionManager source = (SessionManager)event.getSource();

        // create a new player if a new recvstream is detected
        if (event instanceof NewReceiveStreamEvent)
        {
            String cname = "Java Media Player";
            ReceiveStream stream = null;

            try
            {
                // get a handle over the ReceiveStream
                stream =((NewReceiveStreamEvent)event)
                        .getReceiveStream();

                Participant part = stream.getParticipant();

                if (part != null) cname = part.getCNAME();

                // get a handle over the ReceiveStream datasource
                DataSource dsource = stream.getDataSource();

                // create a player by passing datasource to the
                // Media Manager
                newplayer = Manager.createPlayer(dsource);
                System.out.println("created player " + newplayer);
            } catch (Exception e) {
                System.err.println("NewReceiveStreamEvent exception "
                                    + e.getMessage());
                return;
            }

            if (newplayer == null) return;

            playerlist.addElement(newplayer);
            newplayer.addControllerListener(this);

            // send this player to player GUI
            playerWindow = new RTPPlayerWindow( newplayer, cname);
        }
    }
```

See `RTPUtil` in "RTPUtil" on page 223 for a complete example.

A

JMF Applet

This Java Applet demonstrates proper error checking in a Java Media program. Like `PlayerApplet`, it creates a simple media player with a media event listener.

When this applet is started, it immediately begins to play the media clip. When the end of media is reached, the clip replays from the beginning.

Example A-1: `TypicalPlayerApplet` with error handling. (1 of 5)

```java
import java.applet.Applet;
import java.awt.*;
import java.lang.String;
import java.net.URL;
import java.net.MalformedURLException;
import java.io.IOException;
import javax.media.*;

/**
 * This is a Java Applet that demonstrates how to create a simple
 * media player with a media event listener.  It will play the
 * media clip right away and continuously loop.
 *
 * <!-- Sample HTML
 * <applet code=TypicalPlayerApplet width=320 height=300>
 * <param name=file value="Astrnmy.avi">
 * </applet>
 * -->
 */

public class TypicalPlayerApplet extends Applet implements
ControllerListener
{
    // media player
    Player player = null;
```

173

Example A-1: TypicalPlayerApplet with error handling. (2 of 5)

```
    // component in which video is playing
    Component visualComponent  = null;
    // controls gain, position, start, stop
    Component controlComponent = null;
    // displays progress during download
    Component progressBar       = null;


    /**
     * Read the applet file parameter and create the media
     * player.
     */

    public void init()
    {
      setLayout(new BorderLayout());
      // input file name from html param
      String mediaFile = null;
      // URL for our media file
      URL url = null;
      // URL for doc containing applet
      URL codeBase = getDocumentBase();

      // Get the media filename info.
      // The applet tag should contain the path to the
      // source media file, relative to the html page.

      if ((mediaFile = getParameter("FILE")) == null)
        Fatal("Invalid media file parameter");
      try
      {
        // Create an url from the file name and the url to the
        // document containing this applet.

        if ((url = new URL(codeBase, mediaFile)) == null)
          Fatal("Can't build URL for " + mediaFile);

        // Create an instance of a player for this media
        if ((player = Manager.createPlayer(url)) == null)
          Fatal("Could not create player for "+url);

        // Add ourselves as a listener for player's events
          player.addControllerListener(this);
      }
      catch (MalformedURLException u)
      {
        Fatal("Invalid media file URL!");
      }
```

Example A-1: `TypicalPlayerApplet` with error handling. (3 of 5)

```
        catch(IOException i)
        {
            Fatal("IO exception creating player for "+url);
        }

        // This applet assumes that its start() calls
        // player.start().This causes the player to become
        // Realized. Once Realized, the Applet will get
        // the visual and control panel components and add
        // them to the Applet.  These components are not added
        // during init() because they are long operations that
        // would make us appear unresposive to the user.
    }


    /**
     * Start media file playback.  This function is called the
     * first time that the Applet runs and every
     * time the user re-enters the page.
     */

    public void start()
    {
        // Call start() to prefetch and start the player.

        if (player != null) player.start();
    }


    /**
     * Stop media file playback and release resources before
     * leaving the page.
     */

    public void stop()
    {
        if (player != null)
        {
            player.stop();
            player.deallocate();
        }
    }


    /**
     * This controllerUpdate function must be defined in order
     * to implement a ControllerListener interface.  This
     * function will be called whenever there is a media event.
     */

    public synchronized void controllerUpdate(ControllerEvent event)
    {
        // If we're getting messages from a dead player,
```

Example A-1: `TypicalPlayerApplet` with error handling. (4 of 5)

```java
        // just leave

        if (player == null) return;

        // When the player is Realized, get the visual
        // and control components and add them to the Applet

        if (event instanceof RealizeCompleteEvent)
        {
            if ((visualComponent = player.getVisualComponent()) != null)
                add("Center", visualComponent);
            if ((controlComponent = player.getControlPanelComponent()) != null)
                add("South",controlComponent);
                // force the applet to draw the components
                validate();
        }
        else if (event instanceof CachingControlEvent)
        {

            // Put a progress bar up when downloading starts,
            // take it down when downloading ends.

            CachingControlEvent e = (CachingControlEvent) event;
            CachingControl      cc = e.getCachingControl();
            long cc_progress       = e.getContentProgress();
            long cc_length         = cc.getContentLength();

            // Add the bar if not already there ...

            if (progressBar == null)
                if ((progressBar = cc.getProgressBarComponent()) != null)
                {
                    add("North", progressBar);
                    validate();
                }

            // Remove bar when finished ownloading
            if (progressBar != null)
                if (cc_progress == cc_length)
                {
                    remove (progressBar);
                    progressBar = null;
                    validate();
                }
        }
        else if (event instanceof EndOfMediaEvent)
        {
```

Example A-1: `TypicalPlayerApplet` with error handling. (5 of 5)

```
        // We've reached the end of the media; rewind and
        // start over

        player.setMediaTime(new Time(0));
        player.start();
    }
    else if (event instanceof ControllerErrorEvent)
    {
        // Tell TypicalPlayerApplet.start() to call it a day

        player = null;
        Fatal (((ControllerErrorEvent)event).getMessage());
    }
}


void Fatal (String s)
{
    // Applications will make various choices about what
    // to do here.  We print a message and then exit

    System.err.println("FATAL ERROR: " + s);
    throw new Error(s);  // Invoke the uncaught exception
                         // handler System.exit() is another
                         // choice

    }
}
```